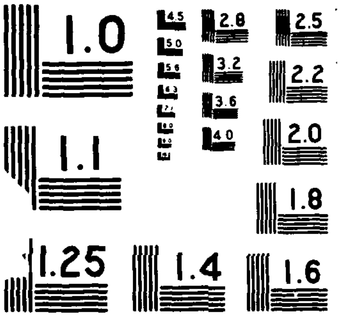
AD-A207 609 IMPLEMENTATION INDICES (1975-1979)VVOLUME 1(U) 1/2
MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER
SCIENCE G M PAPADOPOULOS DEC 88 MIT/CS/TR-432

UNCLASSIFIED F/G 12/6 NL


1.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

4

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution is unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
MIT/LCS/TR-432

5. MONITORING ORGANIZATION REPORT NUMBER(S)
N00014-84-K-0099

6a. NAME OF PERFORMING ORGANIZATION
MIT Laboratory for Computer Science

6b. OFFICE SYMBOL *(If applicable)*

7a. NAME OF MONITORING ORGANIZATION
Office of Naval Research/Department of Navy

6c. ADDRESS *(City, State, and ZIP Code)*
545 Technology Square
Cambridge, MA 02139

7b. ADDRESS *(City, State, and ZIP Code)*
Information Systems Program
Arlington, VA 22217

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
DARPA/DOD

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)*
1400 Wilson Blvd.
Arlington, VA 22217

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Implementation of a General Purpose Dataflow Multiprocessor

12. PERSONAL AUTHOR(S)
Papadopoulos, Gregory Michael

13a. TYPE OF REPORT
Technical

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Year, Month, Day)*
1988 December

15. PAGE COUNT
155

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Tagged Token Dataflow, Dataflow Graphs, Multiprocessor Computer Architecture, Pipelined Processor, Associative Memory (JES)

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

General purpose multiprocessors have largely failed to meet expectations for programmability and performance. We blame the lack of usable parallel programming languages and systems on the underlying processor architecture. Machines built out of conventional sequential processors simply do no support the synchronization demands of parallel execution, so the programmer focuses upon the dangerous and arduous task of discovering a minimum set of synchronization points without introducing nondeterminisim. We argue that processors must be fundamentally changed to execute a parallel machine language, in which parallel activities are coordinated as efficiently as instructions are scheduled.

Dataflow architectures address this challenge by radically reformulating the basic specification of a machine program. These machines directly execute dataflow graphs, which specify only the essential prerequisites for the execution of an instruction--the availability of operands. Unfortunately, dataflow machines, including the M.I.T. Tagged Token

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL
Judy Little, Publications Coordinator

22b. TELEPHONE *(Include Area Code)*
(617) 253-5894

22c. OFFICE SYMBOL

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

*U.S. Government Printing Office: 1985—507-047

19. Dataflow Architecture (TTDA), have labored under a number of implementation burdens, notably the apparent need for a fully associative operand matching store which discovers when instructions are able to execute.

We introduce and develop a novel dataflow architecture, the Explicit Token Store (ETS), which directly executes tagged-token dataflow graphs while correcting a number of inherent inefficiencies of previous dataflow machines. In the ETS model, operand matching is performed at compiler-designated offsets within an activation frame. We show that the ETS is compatible with the TTDA by giving translations from TTDA machine graphs to ETS machine graphs. Finally, we describe an implementation of an ETS dataflow multiprocessor, called Monsoon, now under construction.

Implementation of a General Purpose Dataflow Multiprocessor

Gregory Michael Papadopoulos

MIT / LCS / TR-432
August 1988

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.

Implementation of a General Purpose Dataflow Multiprocessor

Gregory Michael Papadopoulos

Technical Report MIT / LCS / TR-432
August 1988

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02139

Abstract

General purpose multiprocessors have largely failed to meet expectations for programmability and performance. We blame the lack of usable parallel programming languages and systems on the underlying processor architecture. Machines built out of conventional sequential processors simply do not support the synchronization demands of parallel execution, so the programmer focuses upon the dangerous and arduous task of discovering a minimum set of sychronization points without introducing nondeterminisim. We argue that processors must be fundamentally changed to execute a **parallel machine language**, in which parallel activities are coordinated as efficiently as instructions are scheduled.

Dataflow architectures address this challenge by radically reformulating the basic specification of a machine program. These machines *directly execute* dataflow graphs, which specify only the essential prerequisites for the execution of an instruction—the availability of operands. Unfortunately, dataflow machines, including the M.I.T. **Tagged Token Dataflow Architecture (TTDA)**, have labored under a number of implementation burdens, notably the apparent need for a fully associative **operand matching store** which discovers when instructions are able to execute.

We introduce and develop a novel dataflow architecture, the **Explicit Token Store (ETS)**, which directly executes tagged-token dataflow graphs while correcting a number of inherent inefficiencies of previous dataflow machines. In the ETS model, operand matching is performed at compiler-designated offsets within an activation frame. We show that the ETS is compatible with the TTDA by giving translations from TTDA machine graphs to ETS machine graphs. Finally, we describe an implementation of an ETS dataflow multiprocessor, called Monsoon, now under construction.

Key Words and Phrases: Tagged Token Dataflow, Dataflow Graphs, Multiprocessor Computer Architecture, Pipelined Processor, Associative Memory.

Acknowledgments

David Culler co-developed many of the fundamental principles of the explicit token store and has been an endless source of inspiration and criticism. His seminal work on loop bounding was essential to the practicality of the approach.

Professor Arvind taught me the real meaning of computer architecture. There is no substitute for his vision, encouragement and trust.

Professors Jim Miller (Brandeis) and Bert Halstead provided the catalyst for the unification of tags and pointers and the development of a more complete imperative model. Robert Iannucci was a critical sounding board for the architecture throughout its development and was pivotal in establishing machine engineering as a legitimate pursuit in the group. Ken Traub, in addition to writing an absolutely first-class compiler, was especially helpful in sorting out the details and how things "really" work. Professor Nikhil was always available to hear my countless half-baked ideas and provided continuous encouragement. Professor Steve Ward included me on several of his forays to set real world straight, especially the development of the NuBus standard. Richard Soley was my tutor for Lisp machine architectures and helped distill the essential elements of symbolic computation.

Constructing a machine could not be contemplated without the persistent efforts and skills of Dr. Andy Boughton, who has also led the design and development of the Monsoon packet switch. Jack Costanza and Ralph Tiberio are absolute heroes in their ongoing detailed design, simulation and construction of the Monsoon processing element. The research contributions of many other group members played important roles. Notably the numerous simulations and studies conducted by Gino Maa, the demand driven and garbage collection techniques of Steve Heller, the token lifetime experiments of Steve Brobst and the token storage assignment algorithms developed by Serge Plotkin.

Professor Michael Dertouzos has invested a deep trust in my abilities, fostered my ambitions, and given me the freedom and unique opportunities to develop and prove my skills.

Most importantly, nothing could have been possible or complete without the devotion and love of my wife, Elizabeth.

To my parents,
Imogen and Michael Papadopoulos

Contents

List of Figures

Chapter 1

General Purpose Multiprocessing

Across the diverse range of multiprocessor architectures, from small collections of supercomputers to thousands of synchronous single-bit processors, all seem to share one undesirable property: they are hard to use. Programming has taken a giant step *backwards*. The application writer must consider detailed and hard-to-measure interactions of the program with the machine; the resulting codes are difficult to debug [36], are of questionable reliability [44], and are far from portable [30]. Even after arduous work in converting an application for parallel execution, the actual improvement in performance is frequently disappointing. Perhaps we do not yet understand how to express parallelism in a way that is machine independent. Perhaps we need more sophisticated compilers and associated debuggers to better exploit a machine's parallelism. Perhaps we are building the *wrong* machines.

It is a pervasive belief that our lack of real success in general purpose multiprocessing is a software problem. Machine architects adapt sequential processors to the parallel setting by providing an interprocessor communication medium and an *ad hoc* set of synchronization mechanisms, like locks or fetch-and-add. Then the compiler, or worse yet the programmer, is expected to partition the application into tasks that can run in parallel using the supplied synchronization primitives to ensure deterministic behavior. While we certainly share the belief that there *is* a software problem, we are convinced that there are equally serious defects in the underlying machines. There needs to be a fundamental change in processor architecture before we can expect significant progress to be made in the use of multiprocessors. The nature of this change is the deep integration of synchronization into the processor instruction set [12]. The instruction set must constitute

a parallel machine language, in which parallel activities are coordinated as efficiently as instructions are scheduled.

The instruction set of a dataflow machine [6] forms such a parallel machine language. An instruction is executed only when all of its required operands are available. Thus, low-level synchronization is performed for every instruction and at the same rate as instructions are issued. It is easy for a compiler to use this fine grain synchronization to produce code which is highly parallel but deterministic for any runtime mapping of instructions onto processors. While there is consensus that dataflow does expose the maximum amount of parallelism, there is considerable debate surrounding efficiency of the execution mechanism. This criticism centers on three points: (1) the number of instructions executed, (2) the relative power of a dataflow instruction and (3) the cost and complexity of a data driven processor.

Recently there has been significant progress in compiling scientific codes for dataflow machines [7]. Substantial programs written in the high-level language Id [40][39] and compiled for a dataflow machine yield dynamic instruction mixes (*e.g.*, percentage of floating point operations) that are nearly equivalent to the same algorithms compiled from FORTRAN and executing on a reduced-instruction set sequential uniprocessor [20] [8]. Moreover, the dataflow program executes essentially the same number of instructions independent of the number of processors, whereas the parallelization of a program for a conventional multiprocessor invariably incurs non-trivial execution overhead (*e.g.*, synchronizing through barriers, task creation [8]) and typically yields *far less* parallel activity.

But how are we to compare the cost, in terms of processor complexity, of executing a dataflow instruction versus executing an instruction from a sequential stream? First, the operation performed by a dataflow instruction is similar in power to an operation on a conventional load/store machine, *i.e.*, `ADD, MULT, LOAD, STORE, BRANCH` *etc.*[1] The difference lies in the way instructions are scheduled. In the von Neumann model, the operands for an instruction are assumed to be available when the program counter points to the instruction. In a dataflow machine an instruction is *asynchronously* scheduled only when its operands have been produced by some other parallel activities, so a dataflow machine must have an **operand matching** mechanism for detecting when an instruction

[1] Here we are restricting our attention to the instruction set of the M.I.T. Tagged-Token Dataflow Architecture (TTDA).

has its required operands. Several general purpose dataflow machines have been built (*e.g.*, ETL Sigma-1 [26], Manchester Machine [22]) or extensively simulated (*e.g.*, M.I.T. TTDA [14]). But it is clear that these machines are far from commercial practicality[2]. A primary reason for this, and a key criticism of dataflow machines, is the complexity of the operand matching mechanism [21].

Our goal is to discover implementation techniques that improve the cost/performance ratio of dataflow processors. Central to the work presented here is a new approach to operand matching. An **Explicit Token Store (ETS)** machine directly executes dataflow graphs while incorporating a new model of storage. The ETS allows the operand matching storage for the execution of a function invocation to be coalesced into an activation frame which is explicitly managed by the compiler. This enables implementation of the operand matching store with conventional (as opposed to content-addressable) memory technology and permits the realization of well-balanced pipelines.

Although this dissertation focuses on the derivation of the ETS within the realm of dataflow architectures, we take the opportunity by way of this introduction to build a stronger case for machines which support fine grain synchronization. We believe that parallel machine architects eventually will have to apply the same concern for the efficient coordination of parallel activities as they presently do for fast sequential execution within a processor. Only then can we expect significant progress in exploiting parallelism in the general purpose setting.

1.1 Why Fine Grain Synchronization

Most multiprocessors are very bad at managing parallelism. Programmers and compiler writers are painfully aware of this fact. The more finely a program is divided into tasks, the greater the opportunity for parallel execution. However, there is a commensurate increase in the frequency of inter-task communication and associated synchronization. So exposing more parallelism, by way of identifying more tasks, does not obviously make the program run faster. In fact, we claim that is largely *un*profitable to expose most of the latent parallelism in programs *unless* synchronization and task management are as efficient as the scheduling of primitive operations like add and multiply.

[2] The ETL Sigma-1 is the best engineered of the group. Presently, a 128 processor 640 MIPS engineering prototype is now operational at the MITI Electro-Technical Laboratory in Japan.

For a given machine, there is a fundamental tradeoff between the amount of parallelism that is profitable to expose and the overhead of synchronization. Sarkar[46] articulates this tradeoff as the competing contributions of the **ideal parallel execution time,** the amount of time required to execute the program in the absence of overhead, versus the **overhead factor**, the extra work required to schedule and coordinate the tasks. The ideal parallel execution time is multiplied by the overhead factor to yield the **actual parallel execution time,** the amount of time required to complete the problem for a given task granularity in the presence of scheduling overhead.

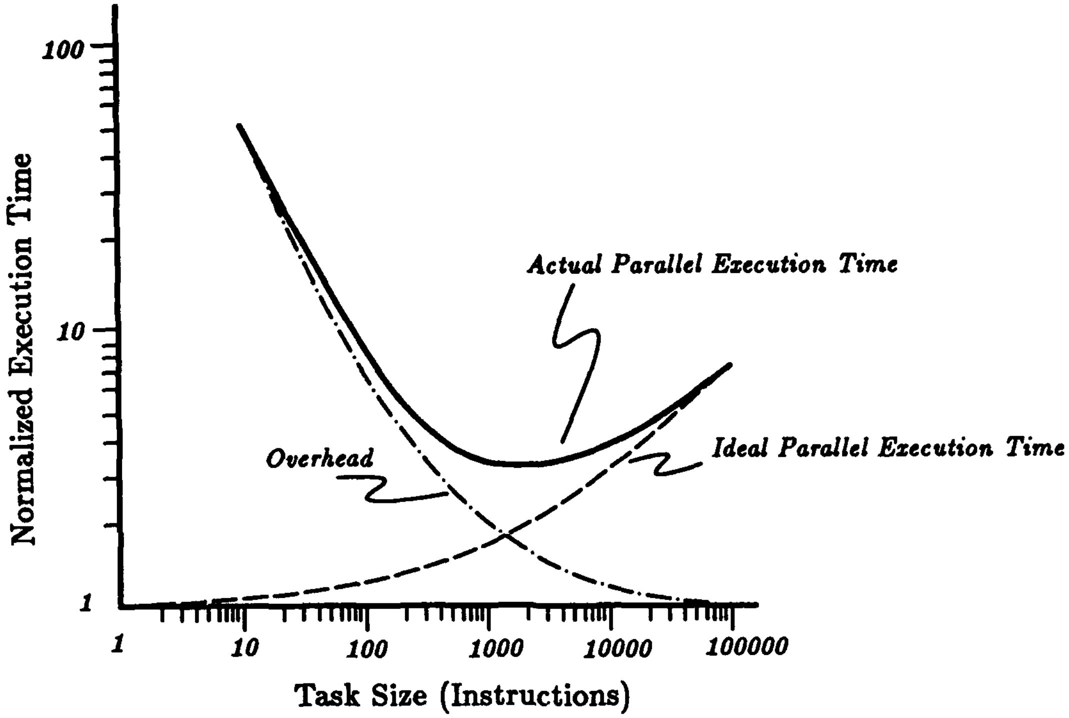
Figure 1.1 illustrates the general characteristic of the parallelism–overhead tradeoff for a typical program running on a machine with ten processors. This plot is suggestive of what was experienced running various programs on contemporary multiprocessors, but it does not express the data from a specific machine or application. The normalized execution time is the ratio n/s where n is the number of processors and s is the actual speedup relative to a single processor. The normalized ideal parallel execution time increases from 1 to n as the task granularity increases from 1, a single instruction, to 100,000 when the entire program executes as a single task. The task overhead factor is given by $(g+o)/g$ where g is the task size in instructions and o is the per-task overhead, in this case 1000 instructions[3].

In this example the actual execution time is a minimum for a task size of about 2,000 instructions yielding a normalized execution time of three—that is, three times slower than the best ideal time. And this plot is optimistic. It is not possible, in general, to partition a given program into tasks of equal size, and the task overhead is a complicated expression that depends upon how the program was partitioned into tasks (*e.g.*, the communication overhead component is a function of the number and sizes of data structures shared by two tasks). Thus, achieving optimal performance in the presence of overhead is much more difficult than simply finding the intersection of the ideal execution and overhead factor curves in Figure 1.1.

1.1.1 Tasks Should be Cheap

Look again at Figure 1.1. We think there is something fundamentally wrong—why should the task overhead be so extraordinarily large? Is it an inherent aspect of parallel compu-

[3] The example employs a per-task overhead of 1,000 instructions which is very *low* as compared to the tasking costs on current commercial multiprocessors and operating systems.



10 Processors, 100000 Instructions, Overhead = 1000 Instructions/Task

Figure 1.1: The Parallelism–Overhead Tradeoff (from Sarkar)

tation or an artifact of the processor architecture used to construct the multiprocessor? We believe the latter. It seems counterproductive to force a programmer or compiler to expend so much effort working around what amounts to a basic deficiency in the processor architecture. If the task overhead were essentially zero, more parallelism would be exposed and exploited and compiling would be far easier. In fact, the entire partitioning and scheduling problem solved by Sarkar for the functional language SISAL [37] would be moot on a machine that had very low task overhead.

By "low overhead" we mean on the order of one to ten instructions, basically *orders of magnitude* better than contemporary multiprocessors. We think the parallelism–overhead tradeoff should look like Figure 1.2. In this ideal world, task overhead is not a first order issue. Instead, the objective is to expose the maximum amount of parallelism by subdividing the computation as finely as possible.

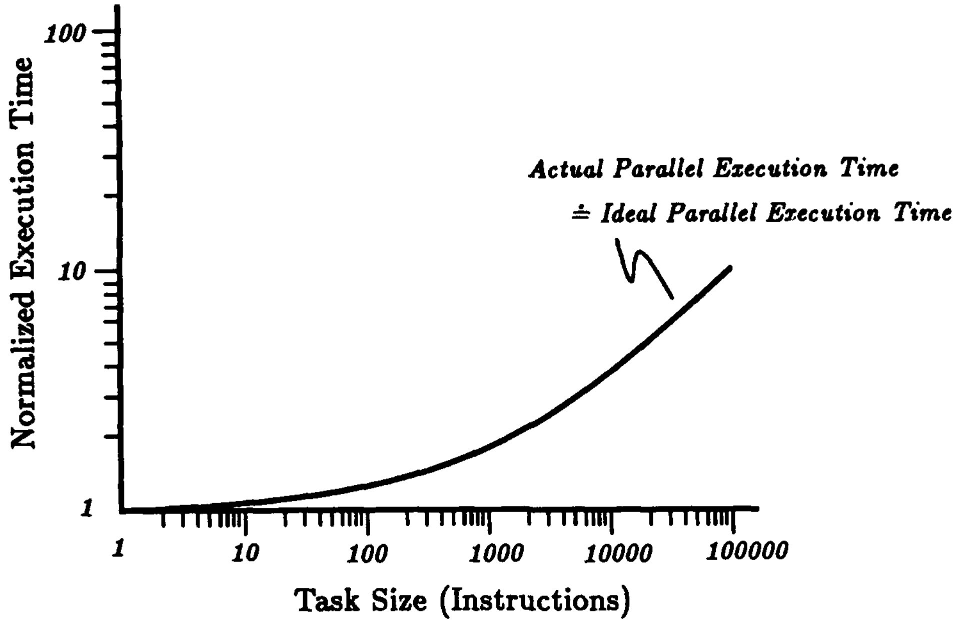


Figure 1.2: Speedup vs. Task Size When Task Overhead is Very Low

1.1.2 A Virtual Memory Analogy

In a way, our argument for hardware support of fine grain synchronization is not unlike the case for architectural changes in order to efficiently implement demand-paged virtual memory. Demand-paging is a convenience for the programmer. It is very tedious for a programmer to manage overlays of code or data, and overlays clash with the semantics of modern programming languages. When data structures get large and access patterns are unpredictable, the programmer using overlays essentially emulates a virtual memory system, with the attendant loss of efficiency—both of the machine and the programmer.

While it is possible to look for the compiler-*forte* that can sift through the access patterns of programs and insert page management code, the problem is so intractable that we have come to insist on hardware support. The essential changes to the hardware are simple to describe, but the effect on the machine architecture is pervasive. Address translation and page fault detection must correctly occur on *every* memory reference, and each instruction that touches memory must be restartable. These two requirements affect almost every aspect of the implementation from cache design to the basic instruction interpretation mechanisms.

Why should the management of complex interactions among concurrently executing entities be any *easier?* At least the demand paging analysis need only focus upon the address patterns of a single locus of control, whereas intertask synchronization analysis must discover the *interaction* of addressing patterns across multiple control loci. We think that a parallel machine should provide, at a minimum, real time synchronization checks on every memory load and store.

We also maintain that compiling for a parallel machine is greatly simplified when the task namespace is "virtualized"; that is, the namespace is much larger than the number of physical processors, so tasks are not bound to limited processor resources like multiple register sets.

An analogy for a machine that supports only a small fixed number of contexts (*cf* the Denelcor HEP [48]) would be a virtual memory system where the hardware supports a small number (say, 64) of simultaneous page translations, *but* where issuing an address that is not presently in the translation buffer causes an unrecoverable error (as opposed to a restartable fault). This puts the burden on the compiler to manage the translation buffer, explicitly inserting and deleting entries. It is not clear that this is any easier than performing demand paging on a machine with *no* hardware support. Similarly, a machine that provides a virtualized task namespace relieves the compiler and runtime system of the tough job of managing a small set of active tasks drawn from the large set of *possibly* active tasks.

In the remainder of this chapter we look more carefully at the overhead incurred in the parallel execution of a program. We build an informal model of parallel computation based upon concurrently executing tasks that can communicate through shared memory. While a conventional processor executes the sequential portion of these tasks very well, it provides little support for moderating the interaction among tasks. Our objective is to sensitize the reader to two of our axioms: (1) a machine should support lots of simultaneous tasks and (2) synchronizing and scheduling these tasks should be very cheap. Not surprisingly, we believe that dataflow machines possess these properties. This dissertation presents a dataflow processor architecture which, we believe, is a simple and appropriate building block for general purpose multiprocessors.

1.1.3 An Informal Task Model of Parallel Computation

We present a model of parallel execution to build our intuition about the cause of high overhead in parallel computation. We do this to motivate architectural support for fine grain synchronization, *not* to put forward an exact, or even complete, performance model for multiprocessors.

We are strongly biased towards programming parallel machines in high level languages which support a dynamic model of storage (*i.e.*, a heap). At present, we do not see how modern languages can be effectively compiled for machines that do not directly supply a uniform address space without forsaking the ability to freely reference shared objects from within the language. In **message passing** machines (*e.g.*, the Cosmic Cube [47] and Intel iPSC) the only way to share data is for the programmer to explicitly code commands to move data from one processor to another, so references to shared objects are emulated by the programmer and are not part of the machine language[4]. We focus instead upon **uniform address space** machines (*e.g.,* the BBN Butterfly [45] , IBM's RP3 [42], Alliant and Cedar [32], Cray-XMP) which provide hardware interpretation of references to addresses which are non-local, meaning there is no requirement for a compile-time distinction between local and non-local storage.

Suppose that we represent an executing parallel program as a **task graph** of interdependent sequential tasks. The nodes of the graph represent activations of sequential tasks, the local storage for which will be contained by an **activation frame**. The edges represent inter-task control and data dependences. We further assume that the execution dependences form a tree, although tasks can also communicate through shared objects on a heap for which references are passed as arguments and results. The shape of the task tree and the connectivity of objects on the heap, in general, cannot be determined statically, as we also permit fully recursive application and dynamic allocation of shared objects. For example, consider the following program which initially invokes `f`, which in turn allocates and returns as a result the object `A`, and makes calls to `g` and `h`;

```
def f() =              def g(A) =              def h(A) =
  { allocate A;          { call g1();            { call h1(A)};
    call g(A);             call g2(A)};
    call h(A);
    return A };
```

We haven't yet taken a position on how `f` is to be computed. Namely, what order is to be imposed on the computation of `g` and `h`? There are roughly three possibilities.

[4] The driving concern is efficiency. It is certainly possible to *emulate* a shared address space on a message passing machine. The inefficiencies of such a scheme should be obvious, notwithstanding the efforts of intensive compile-time analysis.

1. **Sequential.** The familiar sequential order: f calls g and f suspends; g computes, terminates and then continues f; f calls h and suspends; h computes, terminates and continues f; f terminates. These rules are applied recursively to g, h, g1, g2 and h1. So at any time there is a stack of activation frames, of which only the top one is active.

2. **Fringe Parallel.** The calls to g and h are made in parallel: f forks g and h as independent tasks, and f suspends; g and h compute; when *both* g and h have terminated f is continued (*i.e.,* g and h are joined with f); f terminates. These rules are applied recursively to g, h, g1, g2 and h1, but notice that only the leaves of the calling tree are active simultaneously.

3. **Fully Parallel.** A parallel call is made to g and h except that f is not suspended. f terminates after g and h terminate. The recursive application of this rule enables all tasks to run concurrently in the tree of active tasks.

We are not concerned with how the programmer indicates which evaluation method to use, nor whether the programming language semantics are functional or imperative. In either the parallel call or fully parallel case, the task graph unfolds as shown in Figure 1.3. We note that if calls proceed in parallel, there is no way *a priori* to allocate and deallocate local storage for the simultaneous tasks from a stack. In this case, tasks must be given separate activation records which, in general, must be managed more nearly like a heap.

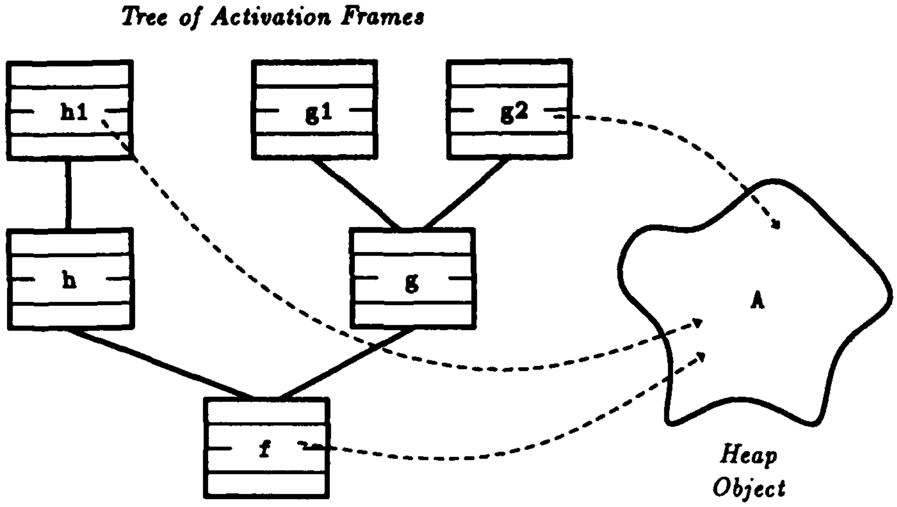


Figure 1.3: Simultaneous Tasks Referencing a Shared Object

The decomposition of the program into parallel tasks is usually accomplished explicitly by the programmer by annotating the source program [30], although functional language (*e.g.*, FP [15], SISAL [37], Id [40][39]) programs can be partitioned automatically by the compiler. There is also considerable effort in the semi-automatic partitioning of programs written in sequential languages, notably FORTRAN, into parallel tasks [3][2].

1.1.4 Parallelism and Synchronization

The opportunity for parallel execution arises by noticing that all tasks whose dependences are satisfied can be freely scheduled. A dependence may either be a parameter to a task (*e.g.*, the pointer to an object), a control prerequisite (*e.g.*, the termination of another task, the acquisition of a resource lock), or dynamic data dependence through the heap (*e.g.*, an element of an array).

The determination of the set of executable tasks is fundamentally a problem of **synchronization,** the act of translating the implicit or explicit assertion of a dependence by one task into a decision either to schedule or block another dependent task. The forms of synchronization required to support various dependences found in parallel programs include the following basic operations:

1. **Producer–Consumer.** A task produces a data structure that is read by another task. If the tasks are executed in parallel, synchronization is needed to avoid the **read-before-write** race.
2. **Forks** and **Joins.** A parallel call forks the thread of control into two tasks which is subsequently joined back together.
3. **Mutual Exclusion.** Concurrent procedures may emit requests that must be processed one at a time, *e.g.*, updating an object or the serialization in the use of a resource.

All forms of synchronization require the *naming* of a synchronization event. At least one bit of state must be associated with this name, indicating whether the event has occurred or is **pending**. When the event occurs, the synchronization can complete. For producer–consumer synchronization this means that a waited-for value has become valid and the consumer can read it; for a join this means that both threads have reached the join and that one may proceed; for mutual exclusion this means that the exclusive resource has been freed for use by a pending requester. We say that a task is **blocked** when it is waiting for a pending event.

The more finely a program is divided into independent tasks, the greater the opportunity for exploiting parallelism; but there is typically a similar increase in the frequency of synchronization. Each pending synchronization event requires a unique name. Thus, as the number of potentially concurrent activities increase, so does the number of simultaneously named synchronization events. If the total number of concurrent tasks are identified dynamically (*e.g.*, parallel execution of nested loops, recursive execution, or even separate compilation) then maximum number of synchronization events is difficult, or impossible, to predict statically. If the synchronization namespace is small, limited for example by the number of registers in a processor, then exposing parallelism is apt to be more difficult as the synchronization namespace must be carefully managed. This requires either static analysis at compile time (tantamount to global register allocation) or fairly expensive runtime (*e.g.*, operating system) management.

Aside from how a synchronization point is named, the most important property of an implementation is how the event is *related to* the completion of synchronization. In an **event driven** system, the event directly causes the blocked task(s) to be scheduled, whereas in a **polled** system the synchronization point is periodically queried by blocked tasks. Event driven systems offer a fixed cost for relating an event to a blocked task, but require a mechanism for *associating* the event to the task (*i.e.*, naming a suspended context). Polling incurs a cost proportional to the amount of time that lapses between the first query of the synchronization point and the occurrence of the event. Systems that rely on polling can degrade rapidly as the number of synchronization events increase.

It is a great challenge to offer *both* a large synchronization namespace *and* to efficiently relate events to waiting tasks. For example, using interrupt vectors to name events can result in the rapid scheduling of blocked tasks but then the synchronization namespace is small and hard to manage. Using words in memory as semaphores permits a synchronization namespace as large as the address space of the machine; but maintaining the relationship between blocked tasks and semaphores can be expensive as it must be *interpreted* by conventional processors.

1.1.5 The Costs of Task State Transitions

At any instant we can assign an **execution state** to every task that reflects its relationship to synchronization events. Figure 1.4 shows a typical task state transition diagram where the states are designated as,

RUN The task is actively executing instructions.

READY The task is able to run but is not currently executing.

WAIT The task is waiting on some event before it can continue executing.

TERM The task has completed all of its computation and has terminated.

Overhead is incurred when the system causes task state transitions. We summarize these costs as follows,

τ_{create} The cost of creating the task, including the cost of allocating storage for the activation.

τ_{switch} The task switching or context switching cost.

τ_{block} The cost of determining that a task must wait for some event.

τ_{sync} The cost of detecting the unblocking event and associating it with the waiting task(s).

τ_{exit} The cost of detecting termination of a task and the reclamation of the task activation storage.

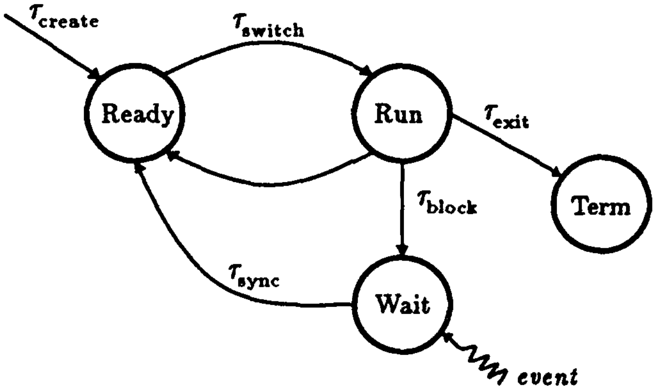


Figure 1.4: A Typical Task State Transition Diagram

In general, the τ_{switch}, τ_{block} and τ_{sync} costs dominate the computation as the task size grows smaller and inter-task communication increases. For example, a shared data structure may have a single locking semaphore to control updates. A task that wishes to read or write the structure must first obtain the lock. If the lock cannot be acquired then the task can either continue polling (*i.e.*, spin-lock) or suspend. Of course, cycles spent polling are strictly overhead but suspending the task incurs a switching cost, $\tau_{\mathbf{switch}}$, and raises the vexing problem of knowing when the lock is free again in order to reschedule the task.

When a problem is decomposed into finer tasks—so as to increase the parallelism—the synchronization costs grow proportionally. A single lock will quickly become a bottleneck, so synchronization must occur on more finely resolved components of a structure, such as the rows of a matrix. This also increases the size of the synchronization namespace.

A conventional processor is good at executing the sequential portions of the individual tasks but it essentially *interprets* the execution of the task graph that describes the parallel execution. Software is forced to maintain the task state (*i.e.*, ready, running, waiting), switch among tasks and, the most costly, enforce the dependences among tasks.

As with most interpreters, a loss of three orders of magnitude over direct execution is often experienced. Indeed, contemporary multiprocessors have task creation and synchronization times in the order of hundreds, thousands, and even tens of thousands of instructions. Hockney [27] has measured the task synchronization overhead on several multiprocessors. For example, he found the task overhead on the LCAP (IBM ECSEC, Rome) running under VM/EPEX was about 10^5 floating-point operation equivalents, and he concluded that tasks would have to have grain sizes from a hundred thousand to a million floating point operations for successful parallel programming on the LCAP. For comparison, Hockney determined the creation and synchronization overhead on the Denelcor HEP [48], undeniably a machine with the best hardware synchronization support of present multiprocessors, ranged from 200 to 800 floating point operations per task.

The interpretation is so expensive that it must be used sparingly, and this is reflected in the programming languages. The parallel extensions of many languages are limited to simple operations (*e.g.*, `fork`, `join`, `do-all`) which are often translated into library and system calls. Because of the cost, these constructs are rarely used in a nested manner and thus lack good compositional properties. If we strip out the sequential computation performed by the tasks, the task control language is weak, incomplete and cumbersome. Moreover, it is difficult to reason about a program's behavior within the language. A programmer must understand the interactions of the parallel constructs with the underlying machine.

1.1.6 Dataflow Machines Directly Execute a "Reduced" Task Graph

A machine executes a parallel machine language when it no longer interprets the task state transitions, but provides direct hardware support to coordinate parallel activities with same efficiency with which it executes instructions. We *do not* mean the translation of the task control into microcode—essentially pushing the interpretation down a level. Instead we search for mechanisms that are simple but readily composed.

A dataflow machine *directly executes* task graphs by taking the following radical positions:

- Every instance of an instruction is a separate task.
- Intertask dependences are expressed purely in terms of simple data dependences on scalars or pointers, and the hardware automatically schedules tasks whose data dependences are satisfied.
- Every location on the heap is a potential synchronization point.

The state transition diagram corresponding to the execution of an instance of dataflow instruction is illustrated in Figure 1.5. A created task waits until the required input data (one ɔr two operands) are available. Once scheduled, the task remains in the RUN state for precisely one instruction, sends its result to all dependent tasks and then terminates. Of course, the transition *costs* have to be low. In fact, we claim that,

$$\tau_{\text{create}} = \tau_{\text{switch}} = \tau_{\text{block}} = \tau_{\text{sync}} = \tau_{\text{exit}} \approx 0.$$

A pipelined dataflow processor can accept a token (data value) every cycle. The execution of a two-input operator (*e.g.*, an add) requires the processing of two tokens: the first token causes a "bubble" while the second token results in the scheduling of the instruction. Thus, for a two-input instruction, we treat the processing of the first token as overhead and assign a $\tau_{\text{block}} = 1$. Because the second token completes the synchronization event *and* executes an instruction we can assign $\tau_{\text{sync}} = 0$. In a pipelined processor a new token (representing a different task) is accepted every cycle, thus task switching occurs on a cycle-by-cycle basis and therefore $\tau_{\text{switch}} = 0$. An instruction creates new tasks by producing result tokens, so τ_{create} is also close to zero[5], whereas $\tau_{\text{exit}} = 0$.

If all instructions were fully synchronous two-input operators (as contrasted with single-input or two-input operators with a constant input) then the execution overhead would be 100% as half of the tokens do not match and cause a τ_{block} of one for each instruction. In practice, more than half of the instructions are single-input or have a constant input and thus incur no task overhead.

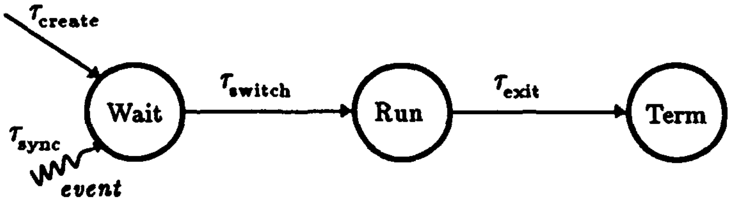


Figure 1.5: Task State Transition of A Dataflow Instruction

Importantly, these "reduced" tasks *naturally compose* into larger aggregates, called **code blocks**. A code block can be allocated, automatically scheduled by the availability of any number of input data, can emit any number of results thereby scheduling other code blocks, and can efficiently issue a termination signal upon the completion of all subordinate tasks. Moreover, this composition leads to the desirable properties of non-strict execution (a code block can begin executing before all of its input data are available, proceeding as far as it can with the available data), exposes and exploits the greatest amount of internal parallelism, and lends itself to hardware structures which can tolerate long pipeline and memory latencies [12].

We should note that these transition costs refer only to instructions within a function activation (code block). Most dataflow machines will incur a measurable function calling overhead, partially attributed to creating a new context for the function. This cost, perhaps ten instructions, has to be amortized over the number of instructions executed by the function body.

[5] This is a function of the number of destinations. For the proposed machine τ_{create} is zero for one or two result tokens, but asymptotically approaches one per result as the number of output tokens grows. See Chapter Four.

1.1.7 Related Work

Denelcor HEP

The Denelcor HEP [48] is one of the very few processors that provide rapid context switching and hardware synchronization. Tasks can be interleaved on a cycle-by-cycle basis, enabling the processor to tolerate long memory latencies. The HEP also provides FULL/EMPTY bits on each register and on each word in the data memory. An instruction attempting to read an EMPTY register behaves like a NO-OP and the instruction is retried the next time the task is scheduled. A fetch of an EMPTY data word causes a busy-wait of the location (performed by a function unit outside the processor pipeline). Thus the HEP has a τ_{switch} and τ_{block} of almost zero, but τ_{sync} that is proportional to the elapsed time until the blocking condition is removed. In addition to an essentially unbounded τ_{sync}, a serious impediment to software development is the limit of one outstanding read per context and a maximum of 64 contexts per processor—the small task namespace has proven too difficult for a compiler to effectively manage.

MASA

The MASA architecture [24], designed to the support MultiLisp [23], tolerates memory latency with context interleaving similar to HEP, but with a virtualized task namespace. MASA is an interesting design in that the *testing for* a blocking condition (basically an EMPTY word on the heap) is essentially free and τ_{block} has been reduced (to the order of tens of instruction times) through heavily optimized exception handling. The unblocking is event driven, and thus fixed cost, where $\tau_{\text{sync}} \approx \tau_{\text{block}}$. In short, MASA "bets on" not being blocked by an attempted read much like a processor assumes that a datum will be cached, only incurring (a relatively large) penalty in the case of a miss. We will be able to judge the effectiveness of this approach only after we have had some experience running substantial programs on the machine.

Dataflow/Neumann Hybrid

The Dataflow/von Neumann Hybrid of Iannucci [28] integrates the fine grain synchronization mechanisms of pure dataflow machines into an efficiently pipelined sequential processor. In the architecture, the tasks are small sequential instruction streams called **synchronization quanta**. The architecture supports a virtualized task namespace which is managed automatically by the hardware. The synchronization costs are such that synchronization quanta should contain about a half-dozen instructions in order to yield acceptable overhead. The hybrid approach is an exciting one as the architect can borrow well-established implementation techniques from high performance von Neumann processors. It is also likely that sequential languages like FORTRAN will be easier to compile for a hybrid—if not by simply ignoring the synchronization features.

For dataflow machines to be ultimately successful, single processor performance will need to be on a par with well-engineered von Neumann machines. A hybrid machine would provide valuable experience to this end. Our present concern is the complexity of a hybrid architecture *vis-à-vis* either a von Neumann or a pure dataflow machine.

1.2 Roadmap

We believe that dataflow machines effectively solve the basic task synchronization problems presented here. We challenge the inference, however, that dataflow machines accomplish this by brute force, and are somehow inherently inefficient, suffer from excessive overhead, or are unmanageably complex. The goal of this thesis is to show that simple data driven processors are attainable, and are appropriate building blocks for general purpose multiprocessors

Chapter Two presents the Tagged-Token Dataflow Architecture as the baseline functionality for our work. We focus upon operand matching as the stumbling block towards an efficient implementation.

Chapter Three formally derives our proposed architecture, the Explicit Token Store (ETS), from a new model of token storage.

Chapter Four shows how to compile TTDA-like dataflow graphs for an ETS by giving a transliteration of the TTDA instruction set. Conventions for I-structure descriptors, closures, and loops are also described.

Chapter Five draws a relationship between an ETS processor and a multithreaded von Neumann machine. We demonstrate how a token can represent a very lightweight thread of a single accumulator machine.

Chapter Six details an implementation of an ETS processor, called Monsoon, which forms the basis of a multiprocessor prototype now under construction.

Chapter 2

The Tagged-Token Dataflow Architecture

The architectural baseline for our work is the M.I.T. Tagged-Token Dataflow Architecture (TTDA), an extensively simulated, general purpose multiprocessor dataflow machine [14] [4] [16] [34]. We insist that any improvements we propose be "compatible" with the TTDA, not in the sense of object-code compatible but that there is a close enough correspondence so we do not require any fundamental change in the way programs are compiled. Nor do we want to distort the relative execution costs to the point where much of the analytic experience with the TTDA is invalidated. By these metrics our proposed architecture, the Explicit Token Store (ETS), is strictly upward compatible with the TTDA as it not only executes TTDA-style dataflow graphs but permits the compiler to imperatively manipulate the token storage.

In this chapter we develop the TTDA as a reference point for the ETS. In particular, we focus upon operand matching as the stumbling block to a cost-effective realization of a tagged-token dataflow machine.

2.1 A TTDA Primer

The tagged-token dataflow architecture is to be contrasted with the popular but inherently limited **static** dataflow machines [19][38][6]. Static dataflow machines are not general purpose multiprocessors as they prohibit the recursive application of a function and preclude functions as first-class objects. The fundamental difference is that a tagged-token machine can dynamically create an instance of a function, whereas a static machine must preplan the storage for all function applications.

A dataflow graph is the parallel machine language of dataflow machines. A node in a dataflow graph is an instruction which executes only when the operands it requires are available. The edges in the graph represent the essential data dependences among instructions. A dataflow graph imposes only a **partial order** of execution; all execution orders which obey the data dependences in the graph yield the same result[1]. This is the source of parallelism in dataflow graphs—a valid execution is to process in parallel all instructions which have their required operands. In contrast, a sequential processor imposes a **total order** on all operations within a task; two operations can occur in parallel only when it can proved that doing so does not change the sequential semantics.

A tagged-token dataflow machine supports simultaneous applications of a function by **tagging** each operand with a context identifier that specifies to which activation it belongs. We call an operand together with its tag a **token**. The tags of two tokens destined to the same instruction within a function must exactly **match**. Each instruction either implicitly or explicitly manipulates tags, and a combination of hardware-enforced rules and compilation disciplines ensures the deterministic behavior of a program. A particular distribution of work or evaluation order might produce *poor performance* but it will never produce *wrong results,* because all evaluation orders are drawn from the partial order specified by the dataflow graph.

2.1.1 A Simple Example

We give here a simple example of a program executing on the TTDA. This exposition is drawn from Arvind and Nikhil [14]. A reader comfortable with the operation of the TTDA may wish to skip to Section 2.2.

Our example program is a function that computes the inner product of two vectors, coded in Id as follows:

[1] We gloss over the problem of a computation that does not terminate. To be precise, equivalence is only guaranteed across execution orders that are **fair**; an enabled operator (one with enough operands to execute) *eventually* must be executed.

```
def vip A B n =
   {s = 0
     in
     {for j from 1 to n
        next s = s + A[j] * B[j]
      finally s}};
```

The dataflow graph for the inner loop of vip is given in Figure 2.1, and Figure 2.2 shows the piece of the graph which computes the expression "s + A[j] * B[j]". The * does not execute (or **fire**) until both of the operands (tokens) are available. Executing an instruction involves consuming all of its input tokens, performing the designated operation, and producing a result token on each of its output arcs.

Figure 2.3 shows a possible firing sequence for the inner loop expression. We say that tokens *flow on the arcs* of the graph, but one should never think of the dataflow graph as representing physical wiring between function modules. The graph is suitably encoded as a linked list of operators and the tokens include a **destination,** the address of the instruction at the end of the arc.

The tokens on the A and B arcs carry only a **descriptor** (or pointer) to the vectors which reside in a separate area called **I-Structure storage.** I-structure storage is similar to conventional directly-addressed storage except that each location in memory is augmented with a *full/empty* bit.

A newly allocated I-structure has all of the presence bits of the corresponding storage locations set to *empty.* When a value is stored into a location, the presence bit for that location is set to *full.* The **select** instruction is a **two-phase** I-structure memory read. The first phase initiates the fetch of the desired location. The second phase comprises the sending of the value of the read location to the destination instruction.

If the presence bit on the location being read is *empty* then the reader is **deferred**—the destination instruction (in the example, *) is remembered by the I-structure storage. When a write to the location finally takes place, the presence bit is set to *full* and all deferred readers for the location are satisfied. Importantly, the two phase nature of select permits reads to be non-blocking: instructions continue to be processed independent of

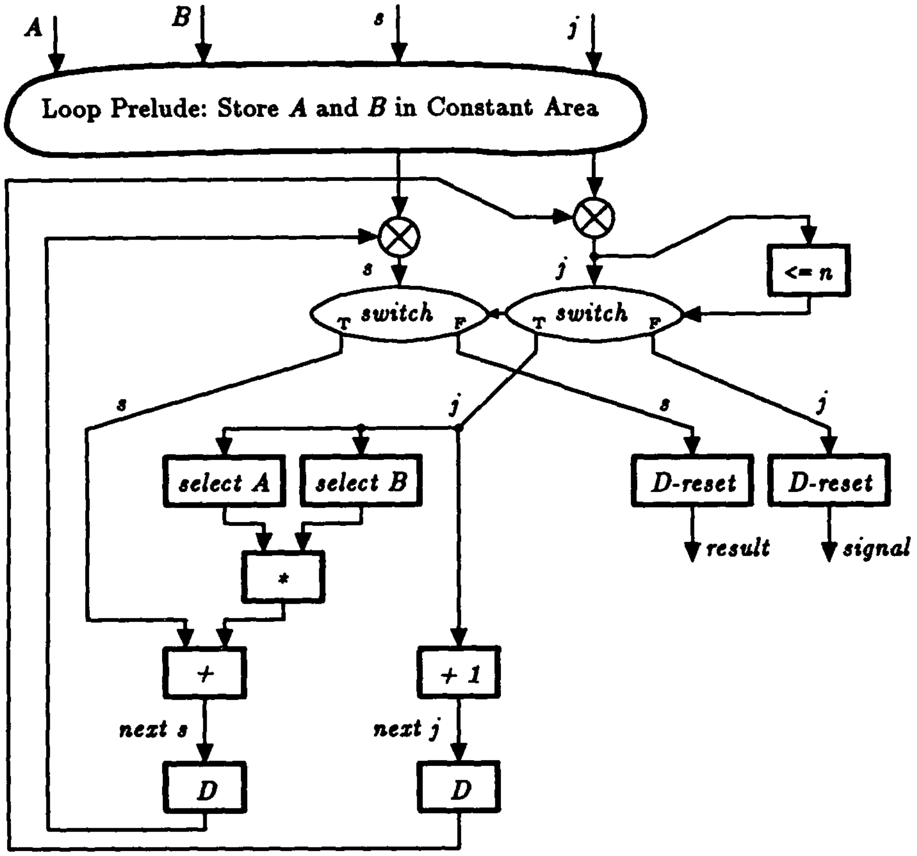


Figure 2.1: A Dataflow Graph for Computing Vector Inner Product (From [15])

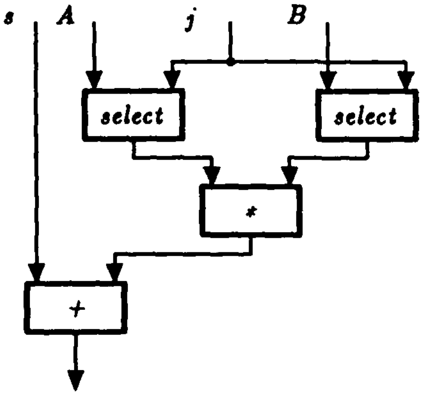


Figure 2.2: Dataflow Graph for "s + A[j] * B[j]"

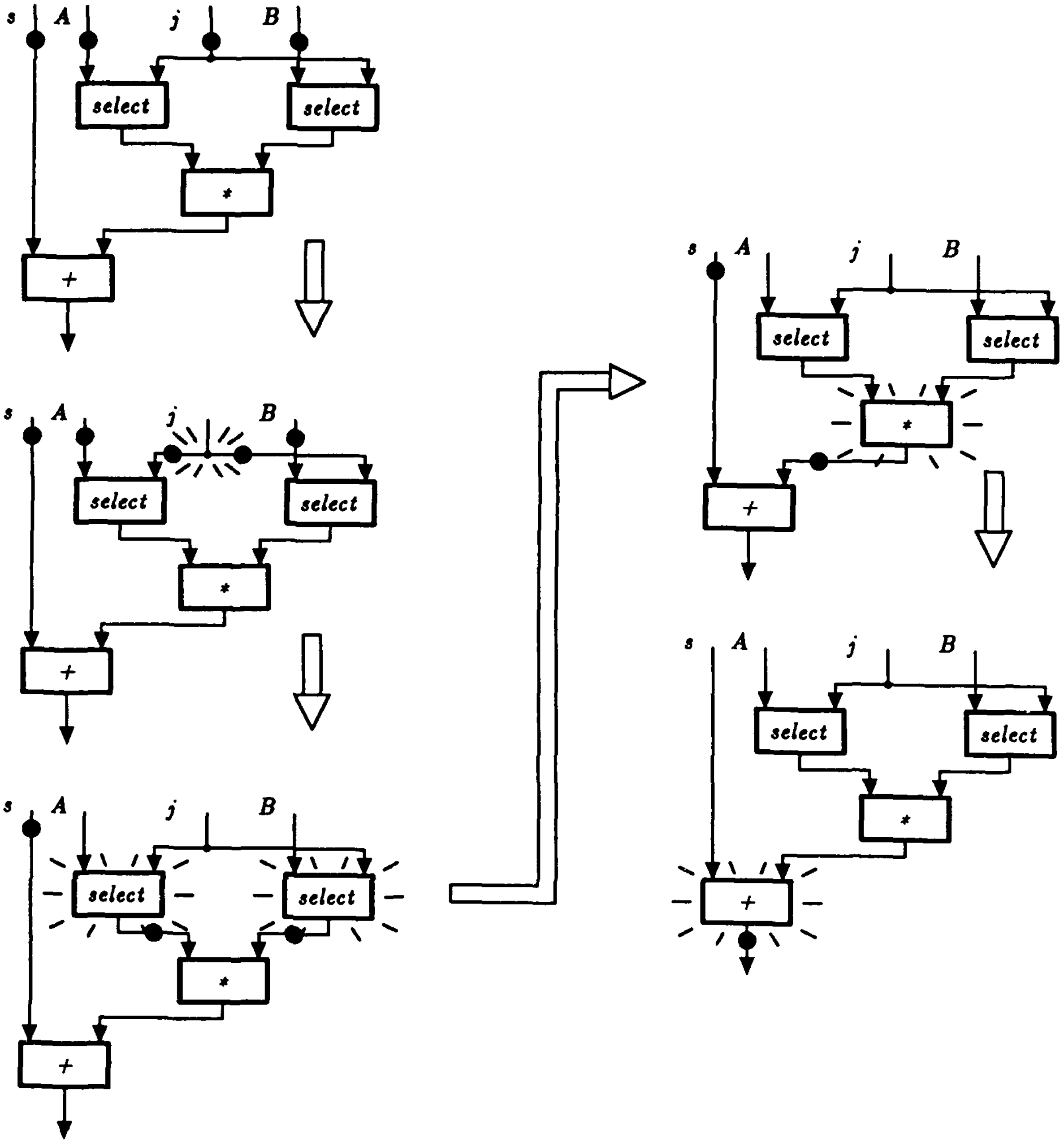


Figure 2.3: A Firing Sequence for "s + A[j] * B[j]" (From [15])

the number of outstanding I-structure reads. Note that I-structure storage is **write-once**: two writes to the same location of a structure is a runtime error. Storage can be reused once a structure becomes garbage and the corresponding I-structure locations have had their presence bits reset to *empty*.

2.1.2 Tags Distinguish Tokens from Different Activations

The D operator sends data from one iteration of the loop to the next. The generation of the indices (the j's) is not dependent upon the computation of the accumulated sum s. Thus all of the iterations of the loop potentially may be operating in parallel. Note that multiple fetches against the elements of A and B may be outstanding even though the next value for s cannot be computed until the previous iteration has completed.

This implies that many * instructions may be enabled simultaneously. How are the different activations of * kept distinct? Tokens destined for each instance of * are distinguished by differing tags. A token looks like,

$$token = \langle c.s_{p}, v \rangle.$$

where v is the **value** and $c.s_p$ is the **tag.** The **statement number** or address of the destination instruction is given by s, and a unique **context** is encoded by c. The **port** p distinguishes a token destined the left (l) versus right (r) input port of a two-input instruction. A combination of hardware and compiler disciplines guarantee that only two tokens will ever have the same tag (modulo the port). The job of the machine is to discover when both tokens have been produced, perform the operation indicated by s, and produce new tokens that encode the destinations of statement s.

Tokens corresponding to many activations of the same instruction s may flow simultaneously through the graph. The hardware implements the following **firing rule** so that tokens from different activations are not confused [14]:

- An instruction is ready to execute when a *matched* set of input tokens exist, *i.e.*, a set of tokens for all of its inputs have the same tag $c.s$.
- The result of the execution is tagged with $c.s'_{p'}$, *i.e.*, the p' port of the destination instruction s' in the same context c.

In order to communicate values to a *different* context it is necessary to manipulate contexts on tokens. This is performed by two instructions: **extract-tag** produces a value similar to the current tag except that s refers to a different statement number (for return address), and **change-tag** is a dyadic operator that takes an arbitrary value and a value of type tag, and produces a token comprising the first value and the new tag. **Change-tag** is the method by which values are **sent** between contexts.

$$\begin{aligned} \texttt{extract-tag:} \quad & \langle c.s,\ \_\rangle \Rightarrow \langle c.s',\ c.s''\rangle \\ \texttt{change-tag:} \quad & \langle c.s_l,\ \hat{c}.\hat{s}_{\hat{p}}\rangle \times \langle c.s_r,\ v\rangle \Rightarrow \langle \hat{c}.\hat{s}_{\hat{p}},\ v\rangle \end{aligned}$$

In the **vip** example, tokens destined to the same * instruction, but belonging to different iterations of the loop, will have the same s. Thus, the tags of tokens belonging to different iterations must have distinct c's. The role of D, a tag manipulation operator, is to generate a unique c field for the next iteration. The alert reader will notice that the two D's are independent (there is no arc between them) yet they must both derive the same new c. We will show how this is accomplished in a moment, but wish to emphasize that consistent tag manipulation is a deeply important hardware characteristic.

In addition, note that the compiler is restricted in the kinds of graphs it can generate. Namely, a collection of instructions that share the same c, called a **code block**, must have the property that, for a given invocation of the code block, a particular instruction can fire at most once. A sufficient condition is that the graph for the code block is acyclic.

2.2 Tag Efficiency

Tags under this rather "pure" tagged-token model will be large and unwieldy because over the execution of a program every function activation and every *iteration* of a loop must be assigned a unique context. This is tantamount to managing a very large address space.

A number of techniques can be employed to reduce the size of a tag and simplify the manipulation rules. These techniques involve adding structure to the tag, changing the compilation of graphs, or both. Generally speaking, the tendency is to replace *logical* components of the tag, namely c, with fields that have *physical* significance. Thus, tag components are tied closer to machine resources. We loosely call the degree of this correlation the **efficiency** of a tag representation. We will explore the following techniques:

1. **Recycling.** Under certain conditions a particular value of c can be safely reused.
2. **Iteration.** By adding structure to c, we permit loops to locally manage a context subspace.
3. **Context Registers.** The c field encodes the name of a register, introducing new processor state.

2.2.1 Recycling Tags

If tags are never recycled and tag manipulation is performed entirely locally, then the c field must be unbounded. This is essentially the **U-interpreter** tagged-token model [10] [9]. Clearly, any realistic implementation will require c to be a fixed size.

Suppose we require an activation that wishes to invoke a procedure to make a request of a **resource manager** in order to get a new context. We'll call this operation **get-context**. While **get-context** is not a purely local operation, its easy to make a distributed resource manager whereby each manager get some large piece of the c namespace. This is similar to dividing up a virtual address space in a conventional multiprogramming environment.

Now we will make two additional demands on the way programs are compiled into dataflow graphs. First, a code block activation must be **self cleaning**, all tokens generated within an activation are ultimately consumed. Second, a code block activation must inform its caller when it has **terminated**, there are *no* tokens remaining for the activation. This recursively implies that all activations created by the called code block have also terminated.

The caller can use the termination signal to generate a **put-context**, which returns the c back to a pool of free names. Now we can reduce the the size of c, making it large enough to name the number of concurrent activities. How large is "large enough"? It is insufficient to provide enough names for the presently active contexts, those contexts that are actually generating work. Instead, there must be enough names for all contexts extant in the calling tree at any instant. If c is not big enough then programs may deadlock and/or the compiler's job is *much* tougher. It must build a data structure corresponding to the call tree and be savvy enough to know which contexts should be allowed to run and which ones should be suspended in order to reuse its context[2].

2.2.2 Iteration Optimization

To keep the different iterations of a loop invocation distinct, each iteration must use a unique context. In order to avoid the excessive overhead of a **get-context** and a **put-context** for each iteration, it is possible to augment the tag with a new field. The iteration field, i, can be managed locally by a loop activation. So a single **get-context** is issued for the loop which essentially returns a *family* of names parameterized by i. A token becomes,

$$token = \langle c.i.s, v \rangle$$

So now the D operator simply increments the i field in order to generate a new context[3]. However, the i field is of fixed size and so D is not a complete solution. Suppose the i field was allowed to wrap back to zero when incremented beyond its capacity. If the first iteration had not yet terminated then there is a very real danger of confusing tokens of the next iteration with ones from the first.

One possible solution is to continue the loop in a fresh context by allocating a new c. In fact, this was the approach of the first Id compiler constructed by Kathail [31]. Another solution, and the current method of choice, is to restrict the number of concurrent iterations to something less than the size of the i field. The loop bounding transformations of Culler and Arvind [17] do precisely this by adding auxiliary dependences such that at most k iterations of a given loop activation can proceed in parallel. In essence, the loop locally manages the recycling of tags. The resulting loop schema is surprisingly efficient and the value k can be set at runtime for each invocation of a loop.

[2] This is our biggest criticism of the Denelcor HEP [48] that only supported 64 simultaneous activations per processor.

[3] We must also provide a **D-reset** that sets the i field back to zero so that results may be communicated to the loop postlude. We assume that all calls to **get-context** return tags with $i = 0$.

At least as important as avoiding the context allocation overhead for each iteration is the ability to share **loop constants** among iterations that have the same c (but a different i). In the `vip` example, `A`, `B` and `n` are compiled as loop constants while only `j` and `s` are passed from iteration to iteration. During the loop initialization (the preamble), loop constants are stored (in *every* processor that is to execute the loop) into a special **constant area** associated with the context c. The loop constant setup charge is quickly amortized over a few iterations of the loop, as references to the constant values are always local to a processor and the per-iteration cost of a **`switch`** and a `D` for an otherwise circulating value can be avoided. In practice, the identification of loops constants significantly reduces the number of circulating loop variables and is an indispensable optimization.

2.2.3 Context Registers

There are still a number of problems even after tag recycling and loop optirnizations are performed. One obvious shortcoming is that s is a global instruction address. Tags can be smaller and code can be dynamically relocated if s instead encodes an *offset* in the code block implied by c.

In order for this to work, the processor must have some way of deriving the base address of the code block from c. Of course, encoding this address directly in c just rearranges fields and the token is apt to become *larger*. The solution employed by the TTDA and the Sigma-1 is to provide a set of **context registers**, addressed by c, that are local state to a processor. The context register contains the code block base address (CBR), so the destination instruction address is readily computed by CBR $+ s$, where now s is an offset.

Other things can be stored in context register as well, notably a pointer to the constant area used by loops to store loop constants. The allocation of c's is now really a problem of managing the *physical resource* of the context registers. Of course, a machine had better provide *a lot* of context registers, enough to name all of the concurrent activations. The requirement of a very large set of context registers presents an engineering challenge almost as great as an efficient implementation of operand matching.

2.3 The TTDA Abstract Pipeline

As tags begin to represent machine resources, it becomes important to supplant the conceptual tagged-token model with an abstraction of a processing element. The TTDA lends itself to a pipelined implementation that directly executes dataflow graphs by asynchronously scheduling instructions on the arrival of data. This pipeline exploits a non-blocking execution of enabled activities, potentially yielding very high performance designs.

As shown in Figure 2.4, an incoming token enters the wait-match section. The incoming tag is compared against the tags of all other tokens currently in the waiting storage. If a match is not found the incoming token is inserted into to waiting storage and nothing is forwarded to the next stage (instruction fetch). If a match is found, the matching token is extracted from the storage and both are forwarded to instruction fetch. Tokens that don't require matching partners (monadic operators) bypass this section.

Tokens that are emitted from the wait-match section are guaranteed not to block in the remainder of the pipeline. In the instruction-fetch stage, the context field c is used as an index into the context registers. Each register encodes a base address for the code and a pointer into the constant area associated with the context. The statement number s is added to the code base and an instruction is fetched from the instruction store. The instruction specifies the opcode, a constant (*i.e.,* immediate) or constant offset, and a list of destinations (the diagram shows only two destinations being processed).

If the instruction specifies a constant offset, then this offset is added to the constant area base pointer (from the context register) and the indicated location is fetched from the constant store. Then, the two operands (the values v_l and v_r from the left and right tokens) and the constant are submitted to the ALU. ALU operations are performed simultaneously with destination list processing. New tokens are assembled in the form token stage by concatenating the destination tags from the form tag unit with the result value from the ALU. Finally, a large token queue provides buffering for excess parallelism.

2.3.1 Storage and Names in the TTDA

The tokens that belong to a particular activation are related by similar tags, they all share the same c. These tokens roughly correspond to the local values in the stack

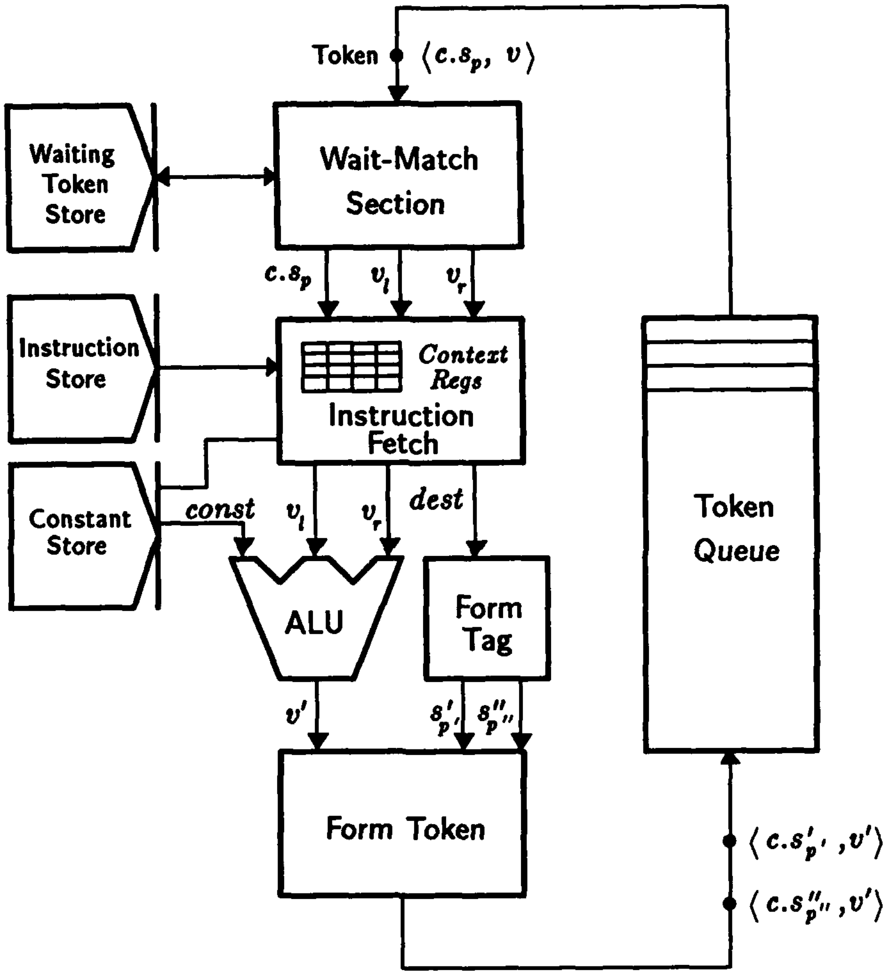


Figure 2.4: The TTDA Token Processing Pipeline

frame of a conventional machine. However, in the TTDA the storage for these values is diffuse—tokens can rest unmatched somewhere in the wait-match section, can occupy the token queue, can be waiting in heap storage for a write to occur, or can be in transit in the interprocessor network. From an engineering perspective, the most troubling implementation issue is the matching store, as it implies content-addressable memory.

Observe that the matching function is only concerned with the structure and *uniqueness* of names (*c.i.s*), and is neither sensitive to their meaning nor the associated value. Our solution (presented in the next chapter) will exploit this property by making an equivalence between a tag and a storage location where the match will take place. This enables a practical realization of the wait-match section while preserving the essential aspects of tagged-token dataflow execution. Before diving into the details of this technique, we first emphasize the importance of an efficient solution to the wait-match section.

2.3.2 The Waiting-Matching Problem

A clear virtue of the dataflow scheduling mechanism is the non-blocking pipeline once an activity has been enabled. However, a fundamental problem arises when trying to implement the wait-match section. The TTDA tag semantics imply an associative search of all tokens in the section against the incoming candidate. Capacious, fully associative memories are not readily realizable so alternatives must be found that are consistent with a goal of single cycle pipeline throughput.

Not only must the wait-match section have a relatively large token capacity [5], but to balance the pipeline properly, it must also have a throughput up to *twice* that of the ALU. An efficient implementation of the wait-match section is central to the goal of translating the TTDA into practical hardware.

If a graph consists mainly of dyadic operators, then most instructions require two inputs and, on average, produce two destinations. The destinations of an instruction are processed independently by the wait-match section. So the ALU can consume and produce two values in a single time step, but the wait-match section takes two time steps to consume two values. Similarly, the wait-match section, on average, can produce only one value per time step. In this scenario the ALU is *at best* fifty percent utilized. In practice we see about seventy percent utilization due to monadic operators, operators with a constant or literal input, and the large number of identities used in termination trees.[4]

There are three approaches to solving this imbalance. The first, and easiest, is to accept it and assign it to copying overhead inherent in data-driven computation. Second, make a wait-match section that runs twice as fast as the ALU. If the processor is truly technological scalable (*i.e.,* no pipeline stage requires "hot" technology, like ECL memory in a CMOS design, relative to other stages), this is really achieved by *slowing down* the ALU to half of the wait-match speed. Finally, we can look for wait-match structures that have internal parallelism—in essence, multiple or interleaved wait-match sections.

This problem might seem a bit contrived, but the reality is many times *worse.* Although fully associative wait-match sections are possible, the size requirement has led to the use of hashing schemes in almost all designs. Even the very best hashing function requires the following steps for a wait/extract operation:

1. **Hash and Compare.** Hash the incoming tag, read the hashed-to location.

2. **Insert.** If the location is not occupied, then insert (write)

3. **Extract.** If the location is occupied then, extract (write empty)

It appears that even this ideal hashing scheme takes two cycles and is difficult, if not impossible, to fully pipeline. As soon as collision logic is employed (this complicates the extract), only the very best of implementations can boast three-cycle performance [26]. In this case a pipelined ALU with a single cycle throughput will be utilized less than twenty percent of the time.

Our proposed solution does *not* solve the pipeline imbalance; the ALU still has twice the throughput of the wait-match section. It does, however, enable single-cycle throughput of the matching function and we believe that the solution readily extends to an interleaved wait-match section.

[4] It might be (convincingly) argued that identities shouldn't be counted as ALU operations since they are strictly overhead. The fact is they consume ALU cycles so for the purpose of pipe balancing a lot of identities really do help. Better compiler technology ought to reduce the number of identities, though, so we head back to fifty percent.

Chapter 3

The Explicit Token Store

The implementation of the token matching function is the bane of practical tagged-token dataflow processors. Tagged-token dataflow execution requires a token to be checked against *all other* waiting tokens for a match. If a match is not found then the token must be remembered by the waiting token storage. If a match is found then the matching token must be extracted from the waiting token storage. The matching function and associated storage is most often realized with a large hashtable and sophisticated insertion and extraction algorithms [26]. The resulting hardware is highly complex, slow and is difficult, if not impossible, to fully pipeline.

Engineering aside, associative memory is hard to manage. For example, suppose that a constant or "sticky" firing rule is desired whereby tokens match as usual, but the matching token is not extracted from the waiting storage. The hashing functions have to be modified to incorporate the new matching rule, and these directions have to be encoded into the tag as well. (Both the Manchester Machine [22] and the ETL Sigma-1 [26] support elaborate matching modes.) A big problem is clean-up. Upon termination of an activation, each constant has to be removed one at a time by sending a "killer" token that *does* cause an extract.

Perhaps there are ways to both pare down the number of tokens that must be compared for a match while also permitting much better control over the waiting token storage. A hint comes from the way that graphs are mapped onto multiprocessor dataflow machines[1]. The tags for these machines are extended to include a processing element

[1] For the Manchester Machine, by "multiprocessor" we mean the "multi-ring" machine in which there are several waiting-matching sections.

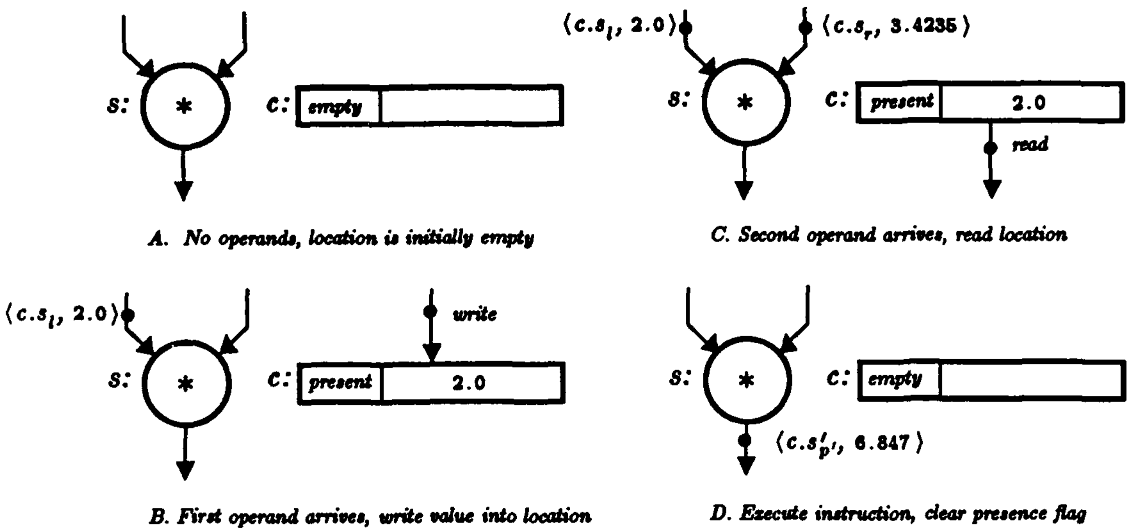
number, or PE field, and there is a separate waiting-matching section for each processor. The matching rules are still the same—the input tags corresponding to the same activation of an instruction must be identical, modulo the port number. And a similar set of compiler disciplines and hardware conventions maintain the uniqueness of tags. Since the partner for a token has an identical tag, it must have an identical PE field. The hardware routes all tokens with the same PE to the waiting-matching section of processor number PE. It is therefore sufficient to check a token against only those tokens on the same processing element. A tag explicitly identifies a *place,* a particular processor, where the match is to occur.

Why not extend the idea to its natural limit? Suppose that a tag $(c.s)$ encodes the address of a unique, global location where the two tokens will meet. That is, the tag context field, c, is the address of an *explicit* rendezvous point.

It is easy to design an explicit waiting-matching function for a two-input operator. As shown in Figure 3.1, every location that corresponds to an activation of a two-input operator is augmented with a **presence bit** that is initially in the *empty* state. When the first token arrives, it notices the *empty* state, and writes its value, v, into the location. The presence bit is then set into the *present* state. The second token notices the *present* state and reads the location, and then sets the presence bit back to *empty.* The operation defined by the instruction s (from the tag) is performed on the incoming token's value and the value read, and a new token is generated with the resulting value, the same c, but a different s.

Note that the location need store only a token's value. The first token's tag may be discarded because the second token will have the same tag. A fully associative or hash table implementation would have to store the first token's tag (or some relatively large fraction of it). In addition, the matching function requires only single port access to the matching memory, meaning that a location is *either* read *or* written during each step. The presence bits, however, undergo a read-modify-write during a step, and there are provisions to perform atomic exchange operations on the value during a step.

The matching operation is also self-cleaning: the location may be used again by another activation as long as certain constraints are met. These constraints basically require a guarantee of mutual exclusivity—if one instruction in the graph has a token on either input then the inputs of all other instructions which specifiy the same location must be (under all possible execution orders) free of tokens (see Chapter Four, Section 4.7).



$\langle c.s_l, 2.0\rangle$
$\langle c.s_r, 3.4235\rangle$
s:
*
c:
empty
present
2.0
read
write
$\langle c.s'_{p'}, 6.847\rangle$

A. No operands, location is initially empty

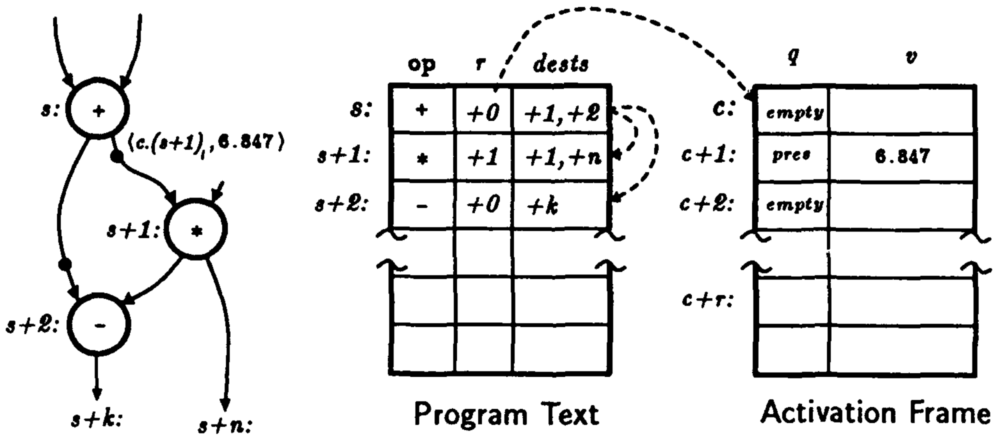
C. Second operand arrives, read location

B. First operand arrives, write value into location

D. Execute instruction, clear presence flag

Figure 3.1: Explicit Matching Operation

Now suppose that all of the locations required for the invocation of a function (or, more generally, a code block) are collected together in a set of contiguous addresses called an **activation frame**. Rather than having the tag point to an exact location where the match will take place, it will encode the *base* address of a frame. The destination instruction supplies the *offset* (r) within the frame for the match. The unique rendezvous location is given by $c + r$. Tokens belonging to different activations of the same function would have tags that encoded different activation frame base addresses (*i.e.*, different c's), but the same code could be shared among all activations.



	op	r	dests
s:	+	+0	+1,+2
s+1:	*	+1	+1,+n
s+2:	-	+0	+k

Program Text

	q	v
c:	empty	
c+1:	pres	6.847
c+2:	empty	
c+r:		

Activation Frame

Figure 3.2: Compilation of a Simple Expression for an ETS

As an example, the compilation of a simple expression is given in Figure 3.2. The dataflow graph is readily encoded as a table whose entries contain the opcode, the activation frame offset r, and destinations encoded as *increments* (for code relocatability) to a tag's s field. For instance, the **+** operator at statement s has two outputs: the $*$ at statement $s+1$ and the **-** at statement $s+2$. Thus the destinations for the **+** operator are encoded as +1 and +2, respectively. The instruction must also encode the port number (left or right) of a destination instruction. We have omitted it in this diagram for simplicity. The **+** operator also specifies the offset in the activation frame, in this case offset +0, where the match will take place. Notice that the **-** operator can re-use this location, as neither input to the **-** can arrive until after the **+** has executed.

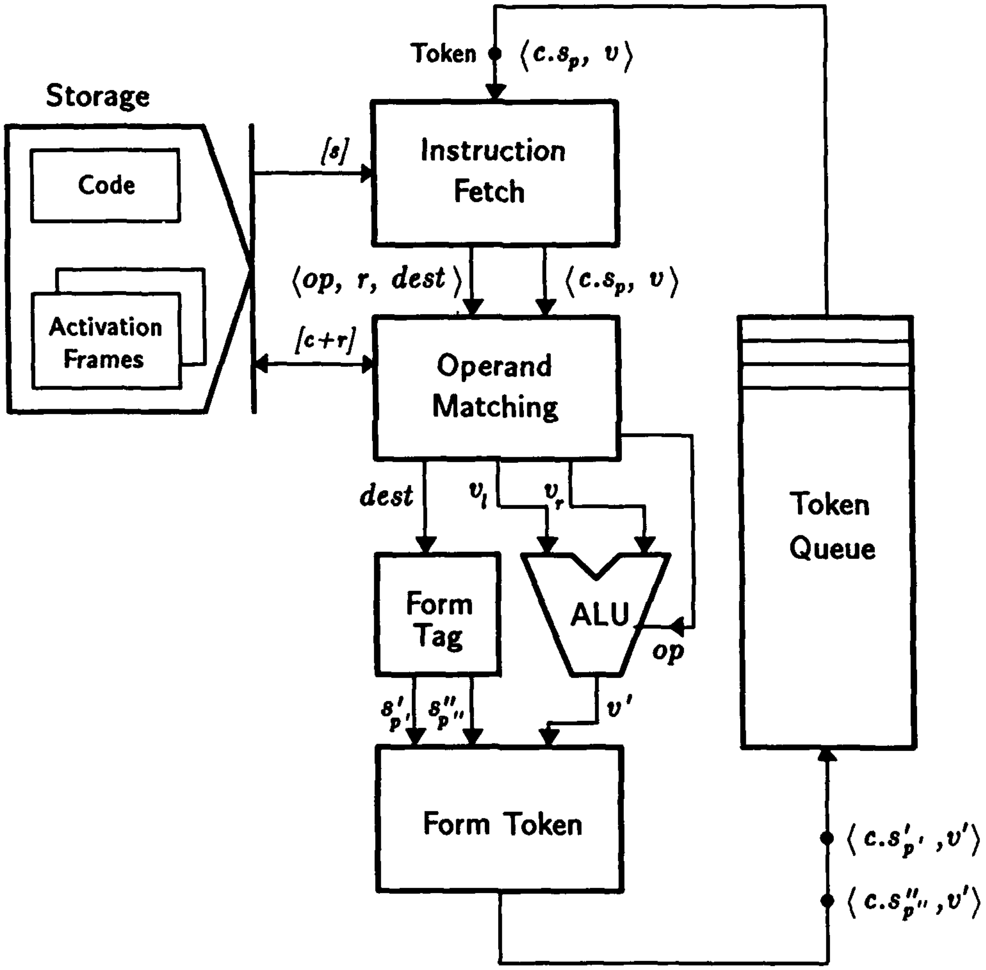


Figure 3.3: An Example ETS Pipeline

Figure 3.3 shows an example pipelined implementation of an ETS processor. Unlike the TTDA pipeline, instruction fetch is performed *before* operand matching. This is necessary because the instruction encodes the offset, r, of the rendezvous location in the activation frame pointed to by the incoming token's c field. The operand matching stage queries the presence bits on location $c+r$. If the slot is empty, the token's value is written into the slot and a "bubble" or no-operation is submitted to the ALU. On the next step the bubble propagates to the form token stage which, in turn, does not insert any tokens into the token queue.

If the slot is full, the operand from the slot is extracted, and this operand along with the value on the token being processed are submitted to the ALU. The destination tags are computed, in the form tag stage, in parallel with the ALU operation. Finally, new token(s) are constructed in the form token stage, which concatenates the destinations from the form tag unit and the result of the ALU. The new tokens are inserted into the token queue.

It certainly appears that an explicit matching function would offer measurable implementation advantages over a large fully associative memory or hashtable. The challenge is to prove that the essential aspects of tagged-token dataflow have been preserved, that nothing has been lost by equating tags to memory locations. We'll form such a proof in two steps. This chapter will complete the picture by developing a somewhat formal model of the explicit token store (ETS). The next chapter will show how to compile *TTDA-like dataflow graphs for an ETS processor.*

3.1 Storage, Tokens and Instructions

The central feature of the ETS is a storage model. Very simply, there is a linear, global address space which is used for instruction, local activation and heap storage. This is just like the von Neumann storage model except that each location is supplemented with a small number of presence bits.

Definition 3.1 *The* **storage** *for an ETS machine is an array of locations, M, such that the i^{th} location, $M[i]$, contains $q.v$ where v is a fixed-size value and q is the* **presence state** *of location $M[i]$.*

The presence state q may be manipulated independently from the location's value v. The manipulation is an atomic read-modify-write where the "modify" is a state transition function to be described in a moment. There are four atomic operations on v: *read,* *write, exchange* and *decrement-exchange.* The latter, used for list building (see Chapter 4, Section 4.3), pre-decrements a value before exchanging it with v.

Tokens are lightweight descriptors of the computation state. Processors transform tokens into new tokens, and perform side-effects on the storage.

Definition 3.2 *An ETS* **token** *is an ordered pair* $\langle c.s_p, v\rangle$ *of a* **tag**, $c.s_p$, *and a* **value**, v, *where,*

c The **context**. A pointer to storage location $M[c]$. By convention, $M[c]$ is the first location in a contiguous sequence of n locations, $M[c], M[c+1], \ldots, M[c+n-1]$, called an **activation frame**. All tokens belonging to the same activation of a code block will have the same c and will be distinguished only by s.

s The **statement number**. A pointer to storage location $M[s]$ that contains the destination instruction for the token. Program text is thus in the same address space as activation frames.

p The operand **port number** of the destination instruction. Only one of two values are possible, l for the left port, and r for the right port. By convention, monadic (single input) operators have only an l port.

v A *fixed-size* value that can also represent a tag. That is, the size of v, in bits, must be at least as large as the size of $c.s_p$. Furthermore, the value-part of a storage location $q.v$ is the same size as a token's tag.

An ETS token looks almost identical to a TTDA token. However, all of the tag components now have physical significance as c and s explicitly refer to storage locations. Another difference is that v is fixed-size under ETS. The model depends upon the ability to freely store a token's value into a memory location, and memory locations e fixed-size.

The processing of a token causes the execution of an instruction. An instruction is pointed to by s and independently specifies the frame offset of the rendezvous point, how a token is to be matched, the ALU operation to be performed in the case of a match, and how a new token is to be formed.

Definition 3.3 *An ETS* **instruction** *is a tuple* $\langle E.r,\ W,\ A,\ T.d\rangle$ *where (See Figure 3.4),*

$E.r$ Specifies the **effective address** for the storage location on which the matching rule will operate.

W Specifies the **matching rule** for the token. W denotes a state transition function applied to a storage location's presence bits. Also as a function of the current state, W specifies the operation on a storage location's value (*e.g.*, read, write or exchange).

A Specifies the **ALU operation** to perform on two values. One value is from the token (v) and the other is from the location specified by $E.r$, for example $M[c+r]$. Monadic instructions operate on only one of these.

$T.d$ Specifies the **token forming rule** for the new tokens resulting from the execution of the instruction. An instruction can have one or two destinations as encoded by $d = \langle s1.p',\ s2.p''\rangle$. The tag for the first destination is $c.s'_{p'}$ where $s' = s + s1$. The second destination is $c.s''_{p''}$ where $s'' = s + s2$.

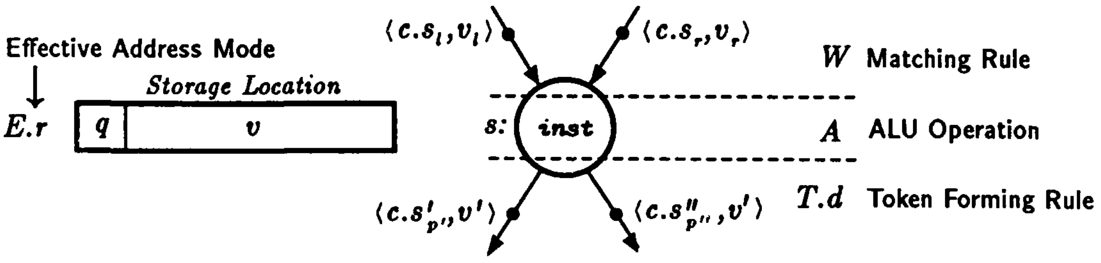


Figure 3.4: ETS Instruction Components

Unlike TTDA instructions, an ETS instruction can encode at most two destinations. This reflects our belief that instructions should all be of the same fixed size, but is otherwise not a necessary feature of the model[2]. Also notice that we do not include any "immediate" or "literal" fields in the instruction, for the same reason. A schematic relationship of storage, tokens, and instructions is given in Figure 3.5.

[2] It is certainly possible to devise macro instruction encoding schemes similar to those employed by complex instruction set computers (CISC).

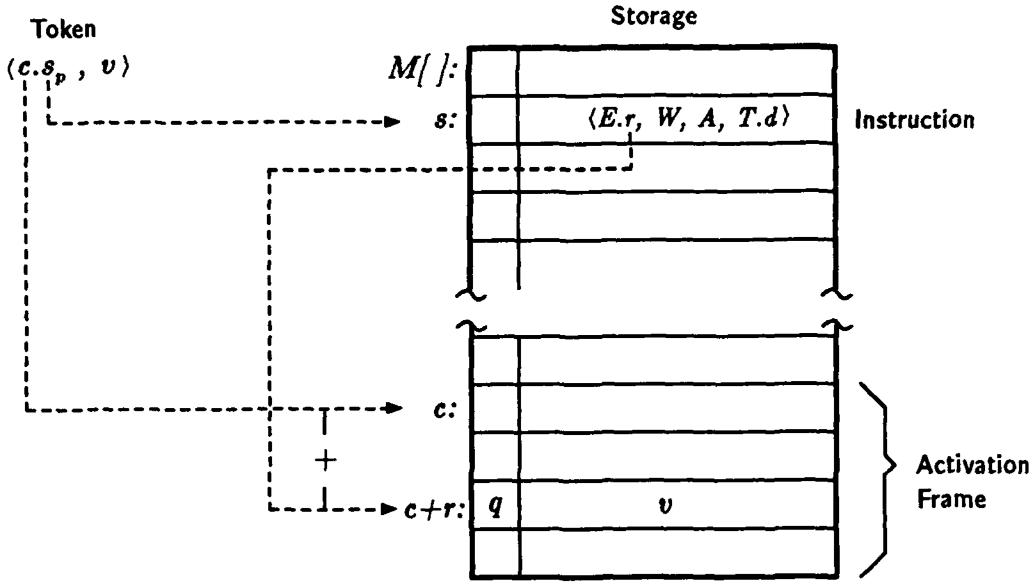


Figure 3.5: Schematic Relationship Between Storage, Tokens and Instructions

3.2 Code Blocks

Just as a collection of von Neumann instructions form a linear sequence, a collection of ETS instructions form a code block. The von Neumann model requires that the instructions appear to execute in their linear sequential order, meaning that instruction $i + 1$ can rely upon the side-effects of instruction i. An ETS code block specifies only a partial ordering expressed solely by the explicit data dependences among the instructions in the code block.

The correct execution of a code block involves a division of responsibility between the compiler and the hardware. The hardware selects a particular order of execution from within the partial order; it need only obey the requirement that an instruction is executed within a finite amount of time after the data it requires are available. The compiler must conform to certain disciplines in order to generate "good" code, so that the mechanisms employed by the hardware still yields a valid execution order. Because the hardware-enforced execution rules are entirely local, a code block is purely a software entity. The same can be said of stack frames on most von Neumann machines—they are strictly a set of conventions, the hardware simply interprets sequences of loads, stores and ALU operations.

Definition 3.4 *An ETS* **code block** *is a digraph* $G = (E, V, I, O)$*, where* $v_j \in V$ *is* *an instruction with a maximum indegree and outdegree of two, and $(v_j, v_k) \in E$ implies $v_j \rightarrow v_k$. $e_i \in I$ is an* **input arc** *that is directed towards v_i (e_i has no source vertex), and $e_o \in O$ is an* **output arc** *that is directed away from v_o (v_o has no destination vertex). See Figure 3.6. It is required that,*

1. G is either acyclic or it can be shown that any order of execution of G that obeys the dependency constraints E will never have more than one token on a given arc.
2. G is **self-cleaning**. When tokens are present on all output arcs in O it must be the case that no tokens are present on the arcs in E or I.

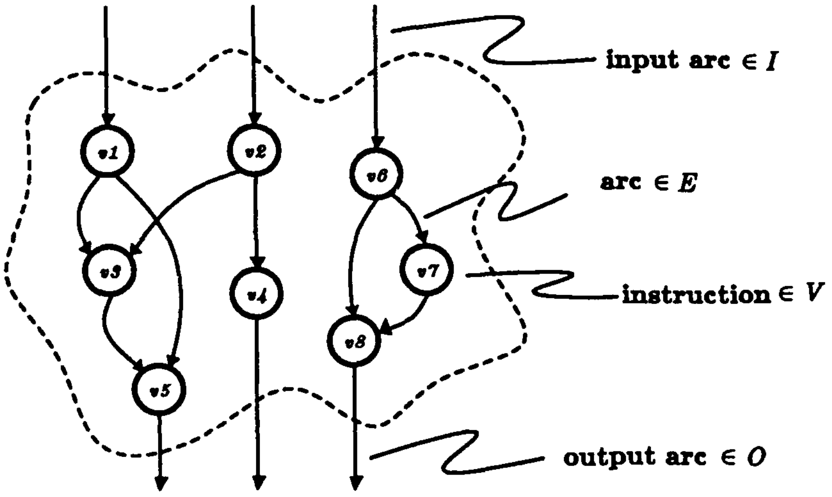


Figure 3.6: An ETS Code Block

The tags of tokens corresponding to the same activation of a code block will have the same context, c, but different statement numbers, s. Tokens belonging to different activations of a code block are distinguished by differing c's. Code blocks are **reentrant**: the instruction text may be shared among any number of simultaneous activations.

The self-cleaning requirement permits the invoker of a code block to reclaim the context (the activation frame based at location c) after it receives all of the result tokens. Meeting this requirement takes some care because code blocks may have instructions that normally do not have data outputs (like a store). Even though it may be known that these instructions have all of their inputs, it is not possible to guarantee that they have executed unless there is a data dependence between the instruction and some code block output. Thus, these instructions must be modified to produce a **signal**, a token of arbitrary value emitted by the instruction when it actually executes. Signals are coalesced by a **signal tree** [49] into a single signal for the whole code block, and this signal is emitted as an output. The existence of the output signal indicates that all zero-output instructions have indeed executed.

The process of invoking a code block involves allocation of an activation frame, sending the arguments to the input arcs, gathering the results from the output arcs, detecting termination and finally reclaiming the activation frame. The hardware does not directly perform any of these tasks. They are performed by a set of compiling conventions that are are in turn built upon the simpler instruction execution mechanisms that form the core ETS model. A set of such conventions are presented in the next chapter. We focus here on the core ETS model, the transformation of input tokens to output tokens.

3.3 Transformation of Tokens

An ETS processor transforms an input token into zero, one, or two output tokens. This transformation occurs in two logical phases: (1) **activity generation** or **matching,** and (2) **token generation.** The matching phase enforces the instruction scheduling prerequisites—the availability of input data. Once an instruction is able to execute, the computation of new values and the creation of result tokens is the responsibility of the token generation phase.

The matching phase performs selective side-effects on storage locations (value and presence bits) and transforms an input token into zero or one activities.

Definition 3.5 *An ETS* **activity** *is a tuple* $\langle c.s, v_l, v_r, A, T.d \rangle$ *where* $c.s$ *is the context and statement of the input token,* v_l *and* v_r *are the operands corresponding to the left and right instruction ports, respectively,* A *is the* ALU *operation to perform and* $T.d$ *is the token forming rule including destination offsets (d).*

Activities are complete, self-contained descriptions. New tokens are generated from activities without any reference to other state, notably storage. From an engineering perspective, this means that activities can be processed by a pipeline that is both non-blocking and hazard-free.

The token generation phase transforms an activity into one or two output tokens. Although the resulting tags often have the same c (but a different s), token forming rules used for argument passing and heap references will replace the tag entirely. Figure 3.7 shows the relationship between these phases.

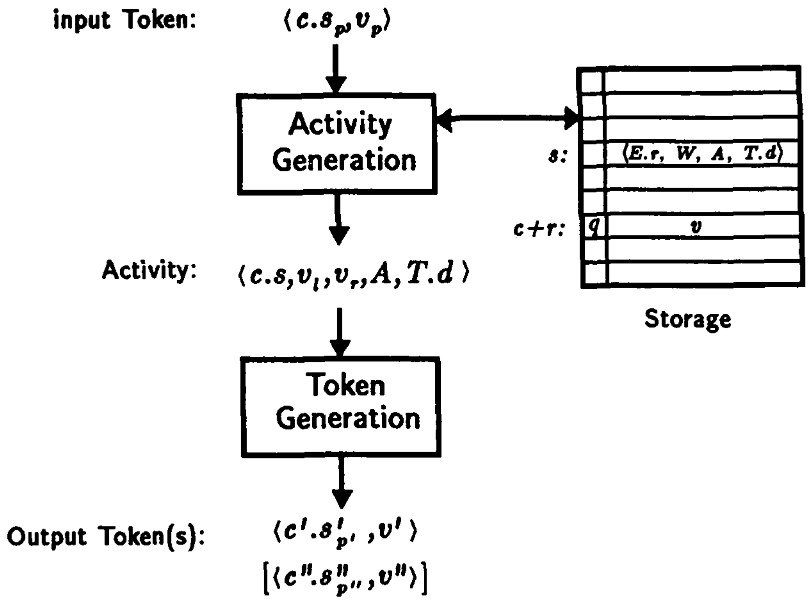

input Token:
$\langle c.s_p, v_p \rangle$
Activity Generation
s:
$\langle E.r,\ W,\ A,\ T.d \rangle$
c+r:
q
v
Storage
Activity:
$\langle c.s, v_l, v_r, A, T.d \rangle$
Token Generation
Output Token(s):
$\langle c'.s'_{p'}, v' \rangle$
$[\langle c''.s''_{p''}, v'' \rangle]$

Figure 3.7: Relationship Between Activity Generation and Token Generation Phases

3.4 Activity Generation

The activity generation (or matching) phase transforms an input token into zero or one activities while also mutating storage. The storage location is determined by $E.r$. The side-effect on the storage location's value and presence state is determined by the matching function W.

Definition 3.6 *The* **effective address** *of the storagc location is generated by one of three effective address modes specified by $E.r$,*

Frame Relative. The location is $M[c+r]$ where c is from the input token's tag, and r is given by the instruction at location $M[s]$, where s is also from the input token's tag.

Code Relative. The location is $M[s + r]$. This is mostly used to access literal constants.

Absolute. The location is $M[r]$. This is mostly used for global constants and resource manager entry points.

Definition 3.7 *A* **matching function** *W is a multivalued state transition function over a port number, p, and a presence state, q, such that*

$$q', m, x = W(q, p)$$

where,

q' The new presence state for the location.

m The operation to perform on the location's value: *read, write, exchange,* or *decrement-exchange.*

x A predicate that can either *inhibit* the generation of an activity or cause it to be *issued.*

The side-effects on a location's state and value are atomic. We now present the token forming rules for dyadic operators, monadic operators and dyadic operators with a constant operand.

3.4.1 Dyadic Operators

The two-input operator where both inputs are non-constant is the basic building block of dataflow graphs. The **firing-rule** for a two-input operator is,

$$\langle c.s_l, v_l\rangle,\ \langle c.s_r, v_r\rangle \ \rightarrow\ \langle c.s, v_l, v_r,\ A,\ T.d\rangle \qquad (3.1)$$

which requires the tags of the two operands to be identical and yields an activity only when both operands are available. This firing rule is implemented by selecting a rendezvous point for the synchronization in the current activation frame: the location $M[c+r]$. The matching function, W_{dyadic} on the associated presence bits is shown in Figure 3.8.

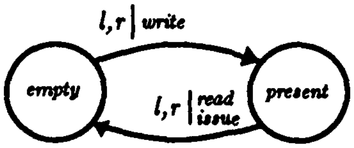


Figure 3.8: The Matching Function for a Dyadic Operator

We annotate transitions on state transition graphs with,

$$\{p\}\left|\begin{matrix} m \\ x \end{matrix}\right.$$

where $\{p\}$ is the set of input ports that cause the transition, m is the operation to be performed on the storage location, and x is given only when an activity should be issued. Assume that the presence state is initially *empty*[3]. Then, the first token to arrive has its value, v_p, written into location $M[c+r]$, the presence state is set to *present* and no activity is generated. The second token to arrive notices the *present* state, reads the matching operand from location $M[c+r]$, resets the presence state to *empty* and issues an activity comprising the incoming tag and value and the matching operand.

3.4.2 Monadic Operators

If an operator has only one input, *e.g.,* **abs**, then the matching function, $W_{monadic}$, is effectively a no-op. The monadic firing rule is, trivially,

$$\langle c.s_l, v\rangle \rightarrow \langle c.s, v, v, A, T.d\rangle \tag{3.2}$$

Input tokens to monadic operators have their port set to left, by convention. The matching function is,

$$W_{monadic}(q, l) = q, read, issue \quad \forall q \tag{3.3}$$

$W_{monadic}$ never changes the presence state or the value of the location, and always issues an activity. The function is undefined for input tokens presented on the right port. We assume the implementation will signal an error whenever W is undefined. This is a catastrophic condition that should occur only if code is incorrectly compiled or there is some hardware failure.

3.4.3 Dyadic Operators with a Constant Operand

A two-input operator will often have a known-present input. This is either a **literal constant**, which is known at compile time, or a **frame constant**, a value that may be shared, without copying, by many activities in a given invocation of a code block. Once a constant has been loaded (literals at load time, frame constants at application time), operations that use the constant *always* generate an activity. The literal firing rule is,

$$\langle c.s_l, v\rangle \rightarrow \langle c.s, v, M[s+r], A, T.d\rangle \tag{3.4}$$

[3] We will always assume that a fresh activation frame has all its presences bits reset to *empty*.

where $M[s+r]$ is the contents of storage location $s+r$. The frame constant firing rule is almost identical,

$$\langle c.s_l, v\rangle \;\rightarrow\; \langle c.s, v, M[c+r],\ A,\ T.d\rangle \tag{3.5}$$

The matching function that implements these firing rules is similar to $W_{monadic}$, except that the presence state must be *present*,

$$W_{read}(\mathit{present}, l) = \mathit{present}, \mathit{read}, \mathit{issue} \tag{3.6}$$

W_{read} leaves the presence bits in the *present* state, reads the location, and always issues an activity comprising the input token's value and the contents of the location. Thus the compilation strategy must guarantee that the location has been initialized before any read can take place.

It is easy to design a matching function that initializes the location with the constant,

$$W_{write!}(\text{don't care}, l) = \mathit{present}, \mathit{write}, \mathit{issue} \tag{3.7}$$

Note that an activity is issued in this case, in effect, a signal that the write has occurred. The equivalent function without acknowledgment inhibits activity generation,

$$W_{nwrite!}(\text{don't care}, l) = \mathit{present}, \mathit{write}, \mathit{inhibit} \tag{3.8}$$

The W_{read} and $W_{nwrite!}$ matching functions can be combined into a single function called a **sticky** match mode. As shown in Figure 3.9, the active operands are presented on the left port and the constant is set by presenting a token on the right port. W_{sticky} permits a single active operand to arrive before the constant is written. Suppose this happens (the active operand arrives before the constant). The active operand is written into the storage location and the location is marked *present*. When the constant arrives, the constant value is exchanged with the waiting operand and an activity is issued. The state is set to *constant* and all subsequent active operands will simply read the location. The W_{sticky} matching function plays an important role in the implementation of I-structures, developed in the next chapter.

3.5 Token Generation

The token generation phase transforms an activity to a result token(s) as specified by the A and $T.d$ parts of the instruction. The token forming rule, $T.d$, specifies how both

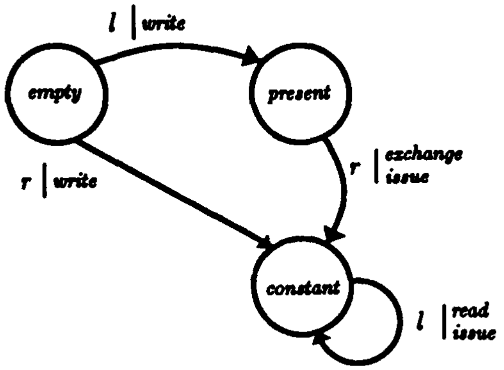
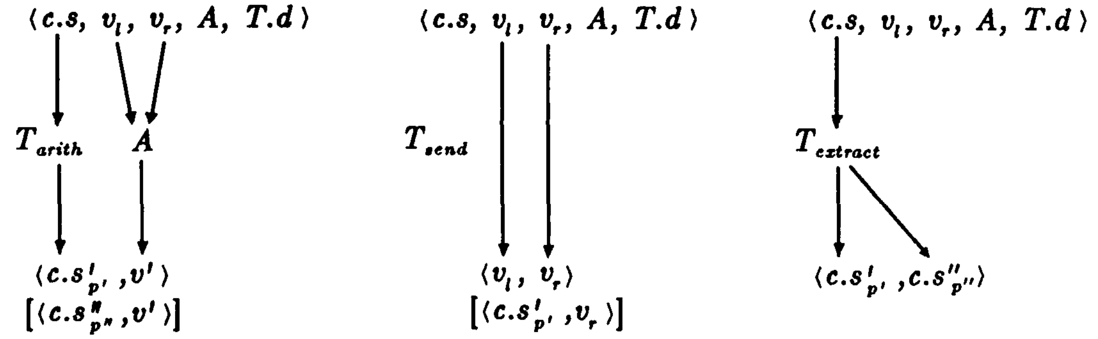


Figure 3.9: The *sticky* Matching Function for a Dyadic Operator with a Constant Input

the new tag and new value are to be formed. Most operations within a code block affect only the statement number, s, of the input tag and produce tokens with the same c and, of course, a new value, v. Other instructions, used for argument passing and heap references, *will* change c, in effect sending a value to a new context. As shown in Figure 3.10, there are three basic token forming rules (the tokens enclosed in braces are optionally produced).

1. The **arithmetic rule** generates the result value by applying A to the two activity values. The result tag is generated from the activity tag by applying simple increments to s and by supplying a new port.
2. The **send rule** generates the result tag from one of the activity values and the result value is the other activity value.
3. The **extract rule** generates the result tag as in the arithmetic rule, but the result value is a modification of the activity tag.


$\langle c.s,\ v_l,\ v_r,\ A,\ T.d\rangle$
T_{arith}
A
$\langle c.s'_{p'}, v'\rangle$
$[\langle c.s''_{p''}, v'\rangle]$
$\langle c.s,\ v_l,\ v_r,\ A,\ T.d\rangle$
T_{send}
$\langle v_l,\ v_r\rangle$
$[\langle c.s'_{p'}, v_r\rangle]$
$\langle c.s,\ v_l,\ v_r,\ A,\ T.d\rangle$
$T_{extract}$
$\langle c.s'_{p'}, c.s''_{p''}\rangle$

Arithmetic Rule

Send Rule

Extract Rule

Figure 3.10: The Arithmetic, Extract and Send Token Forming Rules

3.5.1 The Arithmetic Rule

The arithmetic rule, T_{arith}, is the familiar transformation of an activity into one or two result tokens that have destinations within the same code block as the issuing instruction,

$$\langle c.s, v_l, v_r,\ A,\ T_{arith}.d\rangle \ \rightarrow\ \langle c.s'_{p'}, v'\rangle,\ \langle c.s''_{p''}, v'\rangle \tag{3.9}$$

where the second token, $\langle c.s''_{p''}, v'\rangle$, is optional (*i.e.*, the instruction has two outputs) and,

$$\begin{aligned} s' &= s + s1 \\ s'' &= s + s2 \\ v' &= A(v_l, v_r) \end{aligned}$$

Remember that $s1, s2, p'$ and p'' are encoded by d. We also allow destination tokens to be issued conditionally, permitting the construction of `switch` instructions,

$$\begin{aligned} \langle c.s, v_l,\ \text{TRUE},\ A,\ T_{switch}.d\rangle &\ \rightarrow\ \langle c.s'_{p'}, v_l\rangle \\ \langle c.s, v_l,\ \text{FALSE},\ A,\ T_{switch}.d\rangle &\ \rightarrow\ \langle c.s''_{p''}, v_l\rangle \end{aligned} \tag{3.10}$$

If a TRUE value is presented to the right port (the predicate) of a `switch` then the value from the left token, v_l, is forwarded only to statement s'. If the predicate is FALSE then v_l is forwarded only to statement s''.

3.5.2 The Send Rule

An important property of the ETS model is that tags are pointers into storage and, using boolean and integer operations, a tag can be constructed and manipulated as any other value[4]. For example, a code block could construct a tag whose c field points to a block of unallocated memory (*i.e.*, a free activation frame) and whose s field points to a code block entry point. But how are data to be communicated to the new activation?

[4] An implementation would likely provide special arithmetic unit support to accelerate the manipulation tag data types, however.

The missing operation is the ability to *replace* the current tag with a computed value. This deceptively simple token forming rule is called a **send** operation,

$$\langle c.s, v_l, v_r,\ A,\ T_{send}.d\rangle \ \rightarrow\ \langle v_l,\ v_r\rangle,\ \langle c.s'_{p'}, v_r\rangle \tag{3.11}$$

where v_l is assumed to be of type tag, *i.e.*, $v_l = \hat{c}.\hat{s}_{\hat{p}}$, and the second token is an optional signal. The signal is frequently required for correct termination detection of code blocks. Without it, a **send** instruction looks to the issuing code block like an operator with *no* outputs. Note that **send** is functionally equivalent to the TTDA **change-tag** instruction discussed in Chapter Two.

3.5.3 The Extract Rule

Suppose we are successful in allocating a new activation frame and send the input arguments. In order to get any results back, we must also send *our* tag to the new activation. We need to be able to manipulate a tag derived from the current context. This is accomplished by $T_{extract}$,

$$\langle c.s, x, x,\ A,\ T_{extract}.d\rangle \ \rightarrow\ \langle c.s'_{p'}, c.s''_{p''}\rangle \tag{3.12}$$

Now we can send this tag to the callee, in essence providing our return address. The callee will use this tag to send results back into our context by a convention similar to the one we used to send the input arguments.

3.5.4 Combining Rules

An instruction set drawn from $\{W_{dyadic}, W_{monadic}, W_{sticky}\} \times \{T_{arith}, T_{switch}, T_{send}, T_{extract}\}$ is complete to the extent that we will be able to compile TTDA-style dataflow graphs. An implementation will want to improve the static and dynamic efficiency of the instruction set by optimizing frequently occurring combinations. We will add instructions as we need them, but present a few here for illustrative purposes.

For example, the **inc-s-send** instruction allows an adjustment to the new s field by stealing the $s2$ and p'' fields of d,

$$\langle c.s, \hat{c}.\hat{s}_{\hat{p}}, v_r,\ A,\ T_{inc\text{-}s\text{-}send}.d\rangle \ \rightarrow\ \langle \hat{c}.(\hat{s} + s2)_{p''}, v_r\rangle \tag{3.13}$$

Inc-s-send is used extensively for passing arguments and results, essentially by "indexing" off of the tag $\hat{c}.\hat{s}_{\hat{p}}$. The context to receive the arguments is $\hat{c}$ and the statement $\hat{s}$ is the first statement in a set of entry points. The instruction at statement $\hat{s} + i$ has an input arc on which it expects argument i.

Another common optimization, useful for sending return addresses, is the combination of T_{send} (3.11) and $T_{extract}$ (3.12) to form **extract-send**,

$$\langle c.s, \hat{c}.\hat{s}_{\hat{p}}, v_r, A, T_{extract\text{-}send}.d\rangle \rightarrow \langle \hat{c}.(\hat{s} + s2)_{p''}, c.s'_{p'}\rangle \tag{3.14}$$

$T_{extract\text{-}send}$ sends a tag from within the sending context, $c.s'_{p'}$, as an argument to statement $\hat{s} + s2$ in context $\hat{c}$, which usually remembers the tag and uses it as the base address for returning arguments.

Finally, a combination of T_{send}, $T_{extract}$ and T_{arith} performs an adjustment to $\hat{c}$ and then sends a return address. The T_{fetch} rule is used for reading elements from an array where $\hat{c}$ is the base address and v_r is an index,

$$\langle c.s, \hat{c}.\hat{s}_{\hat{p}}, v_r, inc\text{-}c, T_{fetch}.d\rangle \rightarrow \langle (\hat{c} + v_r).\hat{s}_{p''}, c.s'_{p'}\rangle \tag{3.15}$$

The instruction at $\hat{s}$ could be designed to read location $\hat{c} + v_r$ and send the value to the requester's return address, $c.s'_{p'}$. These *split-transaction* memory operations form the basis for interactions with the heap, as we shall now see. Note that the $A = inc\text{-}c$ is a special ALU operation that adds an integer offset to the context (c) of a tag datatype. (Similarly, the *inc-s* ALU operation adds an integer offset to the statement number).

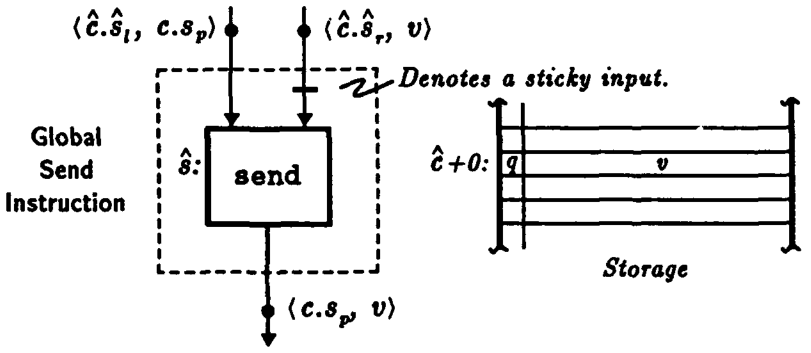
3.6 The Heap

The core ETS model outlined so far is powerful enough to support shared heap objects. So the heap is really a set of software conventions—we can use the same mechanism for the local process of token transformation by an instruction *and* the reading and writing of a global location. As we shall show, a **pointer** to a storage location can be represented with a tag. This should not come as a surprise. A tag, after all, comprises *two* pointers into storage: the frame address c and the instruction address s_p. A tag used as a pointer will encode the storage location using c and encode the operation to perform with s_p.

Consider the code block comprising a single **send** instruction,

$$\langle \textit{frame-relative.0},\ W_{sticky},\ \text{nop},\ T_{send} \rangle$$

(frame relative address, offset $r = 0$, a sticky matching rule, no ALU operation and a send token forming rule). Let $\hat{s}$ be the **send** instruction's address. As shown in Figure 3.11, a token sent to the right port, $\langle \hat{c}.\hat{s}_r, v \rangle$, *writes* its value into location $\hat{c}$. A token sent to the left port, $\langle \hat{c}.\hat{s}_l, c.s_p \rangle$, *reads* the value and returns the result to statement s in context c by forming the output token $\langle c.s_p, v \rangle$.



$\langle \hat{c}.\hat{s}_l,\ c.s_p \rangle$
$\langle \hat{c}.\hat{s}_r,\ v \rangle$
Denotes a sticky input.
Global Send Instruction
$\hat{s}$:
send
$\hat{c}+0$:
q
v
Storage
$\langle c.s_p,\ v \rangle$

Figure 3.11: A Single Instruction Code Block That Performs Reads and Writes on Any Location

We can supply any value for $\hat{c}$ while specifying the same $\hat{s}$. So we need only a single instruction, the **send** at location $\hat{s}$, to allow the reading and writing of *any* storage location. A context that wishes to read a location performs a split-transaction or **two phase** operation. As shown in Figure 3.12, the reading context sends to the location a tag corresponding to an instruction in its context where it wants the result to be sent. The `extract-send` instruction (see equation 3.14) extracts the current tag and creates a new token with this tag as its value, where the new tag is obtained from the input to `extract-send`. When the **send** instruction receives the reader's tag and when the value has also been received, it creates a new token whose destination is back in the reader's code block.

Arrays of locations are readily implemented, where $\hat{c}$ points the base location in an array. Remember that all locations of all arrays can share the same **send** instruction at location $\hat{s}$. A code block can select the j^{th} element of the array by computing $\hat{c} + j$. Note that the `fetch` instruction (equation 3.15) can compute $\hat{c} + j$ and perform the `extract-send` all in the same instruction.

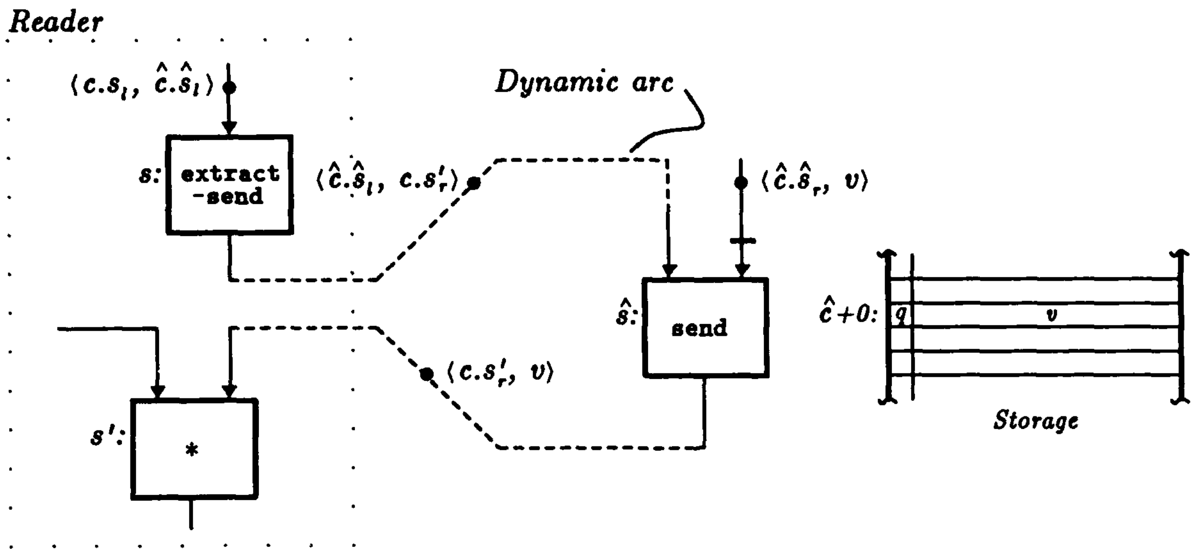


Figure 3.12: A Two Phase Read of a Location

The W_{sticky} matching rule provides a bonus: a single reader can arrive *before* the writer. When the write does finally occur, the waiting reader's tag (presently occupying the location) is exchanged with the value to write. Subsequent reads are processed immediately, as the sticky matching rule leaves the right value in the location. In the next chapter we give a convention for queueing multiple deferred readers of a location.

It is also possible to create a matching function for a send which completely ignores the presence bits associated with the heap location. A write (a token placed on the right port) replaces the current contents, even if the presence state is *present* (*i.e.*, multiple writes do not cause an error). A read of the location always succeeds and returns the current contents, even if the presence state is *empty*. The behavior of such a location is exactly like conventional imperative storage on a von Neumann machine.

To complete the picture, we also require the ability to imperatively set the presence bits of a location into any desired state. But this is easy to do with a suitable W. For example, the i-clear instruction specifies W_{empty}, which always sets the new presence state to *empty*. I-clear can be used by resource managers to reinitialize storage.

3.7 Summary

The explicit token store supplies a new model of token storage which enables a tractable solution to the operand matching problem. The ETS is a strict superset of the TTDA in that it provides TTDA-like functionality while exposing more of the instruction processing mechanism to the compiler. As we shall see in Chapter Five, this enables reasonably efficient compilation of sequential code for an ETS processor. First, we must further support our claim that the ETS subsumes TTDA functionality without sacrifice. We do this in the next chapter by showing how to compile TTDA-like dataflow graphs for an ETS processor.

Chapter 4

Compiling for an ETS Dataflow Processor

The compilation of high level functional languages for the Tagged Token Dataflow Architecture is well-understood [49]. Substantial scientific programs written in the high level language Id and compiled for the TTDA yield dynamic instruction mixes (*e.g.,* percentage of floating point operations) that are within a factor of two of the same algorithms compiled from FORTRAN and executing on a sequential uniprocessor [8].

An important goal of the ETS is compatibility with the TTDA, meaning that the compilation process is nearly the same and the resulting object codes (machine graphs) yield similar dynamic execution characteristics. To meet this goal, we will construct a **transliteration** of TTDA machine graphs into ETS machine graphs. Our principal strategy is to translate individual TTDA instructions *independent* of how the instructions are being composed into larger schemata. This is a safe approach because dataflow instruction execution rules are entirely local (the arrival of data schedules the instruction) and most TTDA instructions do not have side-effects. That is, the semantics of a schema containing a given instruction are unaffected when the instruction is replaced by a graph (possibly comprising a single instruction) which has the same input/output behavior as the original instruction.

For the most part there is a one-to-one correspondence between a TTDA machine instruction and an ETS machine instruction, but there are some notable exceptions:

- **Loop Iteration.** An ETS tag has no iteration component (the i field of a TTDA tag). Iterations of a loop must be assigned different contexts and only bounded loops can be executed efficiently.

- **I-Structure Descriptors.** The ETS has pointers (tags) into I-structure arrays but no I-structure descriptors. (A descriptor encodes the base address of a structure and its upper and lower bounds). This means that bounds information must be stored in memory, for example in an array header.

- **Closures.** The ETS does not have a special closure datatype. Instead, closures have to be encoded as a tag.

- **Three-Input Operators.** Some TTDA instructions can have *three* inputs whenever one of the operands is a literal or loop constant. The same functionality requires two ETS instructions.

- **Instruction Fanout.** An ETS instruction can specify a maximum of two destinations while most TTDA instructions can specify an arbitrary list of destinations. An operator with more than two destinations will require additional `identity` instructions to fanout its result.

- **Slot Assignment.** An ETS instruction must explicitly indicate the slot in an activation frame where a match will take place (*i.e.*, assign r). The compiler must assign the slots to all instructions in a code block such that tokens destined to different instructions do not collide in the same slot.

In this chapter we will resolve these differences by showing how to transliterate the affected TTDA graphs into semantically equivalent ETS graphs. We *do not* show how to compile machine graphs from Id. The interested reader is referred to Traub's definitive treatment of the subject [49].

4.1 Basic Instruction Set Equivalences

In this section we examine individual TTDA instructions in detail and, where possible, give a translation into corresponding ETS instructions. That is, for each TTDA instruction we specify the equivalent ETS instruction components $\langle E.r, W, A, T.d\rangle$ (see Definition 3.3). In the case of TTDA instructions with more than two inputs (*e.g.*, `form-address-i-store`) or more than two destinations, we first rewrite the instruction into an equivalent graph of two-input, two-output TTDA instructions.

4.1.1 Machine Data Types

Before diving into the detailed mapping of opcodes, it is important to understand the relationship between the kinds of values that a TTDA instruction manipulates versus what is provided by an ETS. A **machine data type** is a bit encoding for which the machine provides instructions to manipulate. For instance, integers and floating-point numbers are distinct machine data types if the machine has instructions which can manipulate floating-point numbers. This does not necessarily mean that the machine can *distinguish* among the different types, *e.g.,* hardware type-tags that permit the runtime determination of a machine data type.

Remember that the ETS assumes a fixed-size word (the size of a word in storage and the value-part of a token). All ETS machine data types occupy a single word. The TTDA, however, assumes a much richer set of machine data types, some quite complex. For example, Arvind and Iannucci [11] specify four different integer formats (8, 16, 24 and 32 bits) and two floating point formats (32 and 64 bits). We take the liberty to ignore these distinctions and provide only single-word integers and floating point numbers.

More challenging is the multiplicity of pointer formats—tags, I-structure descriptors, I-structure addresses and closure references—whereas the ETS has no special support for anything but tags. By the end of this chapter we will have shown how the following TTDA machine data types can be implemented by the indicated ETS machine data types.

TTDA Machine Data Types and ETS Equivalents

TTDA Type	ETS Equiv	*Description*
TAG	TAG	Tags used as references to contexts
INT	INT	2's complement integer
FLT	FLT	Floating point number
BITS	BITS	Bit field with boolean TRUE and FALSE values
ISD	TAG	I-structure descriptor
ISA	TAG	I-structure address
CLOSURE	TAG	Closure value

An important property of the ETS is suggested by the above table; *ETS tags and pointers have the same representation.* This unification reduces the number of machine data types (and thus the number of specialized instructions) while creating a richer and more powerful pointer. That is, an ETS tag, $c.s_p$, has *two* independently specifiable pointers: a pointer to an instruction (s_p) and a pointer to a location in storage (c).

4.1.2 TTDA Instructions

A TTDA instruction, as defined by Arvind and Iannucci [11], is somewhat more complex than an ETS instruction. In particular, a TTDA instruction can specify an arbitrary number of destinations and can have up to *three* inputs, as long as one of the inputs is a constant (the token matching operation still works only on pairs of values). The general form of a TTDA instruction is defined as follows (refer to Figure 4.1):

Definition 4.1 *A* **TTDA instruction** *is a tuple* $\langle Op, X.k, D\rangle$ *where,*

Op Specifies the instruction opcode, *e.g.*, `add, switch, i-fetch`.

$X.k$ Specifies the "wiring" of the left operand, right operand and constant (if any) to the three instruction inputs. The constant k is either a **literal** which is encoded directly into the instruction, or a **loop constant**, specified as an *offset* into the constant area associated with the current context. An instruction can have at most one constant input.

D Specifies a list of destination instructions, $\langle s'.p'.n', s''.p''.n'', \ldots\rangle$. A destination, $s.p.n$, encodes the instruction address s, port p and the number of tokens required to enable the instruction n (1 or 2). Most instructions can specify any number of destinations.

There are a few exceptions to the general TTDA instruction format. The `i-fetch` and `change-tag` instructions can specify only a single destination whereas the `switch` instruction can specify *two* destination lists, one for the TRUE branch and one for the FALSE branch.

4.1.3 Rewriting Three-Input Instructions

The TTDA can match only two tokens per instruction but, because of the instruction constant k, an instruction can have three inputs. An ETS instruction can have at most

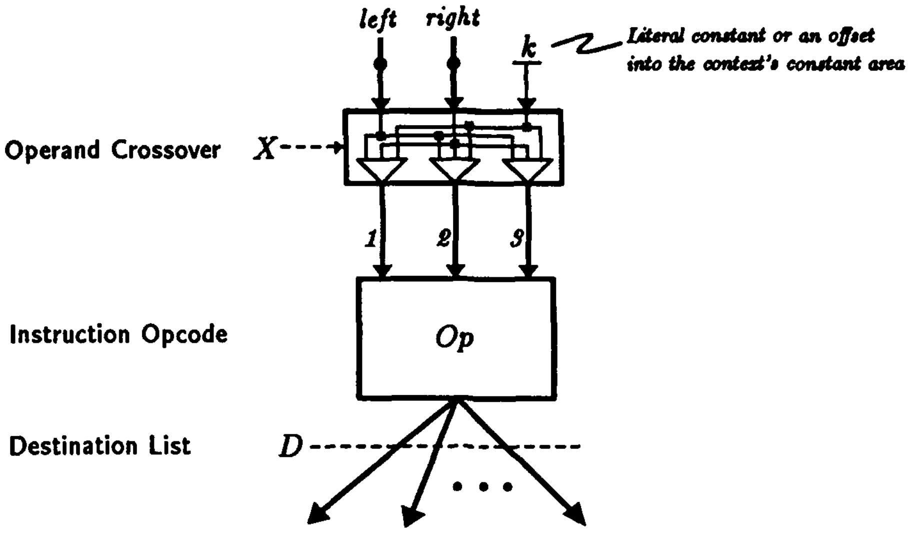


Figure 4.1: TTDA Instruction Components

two inputs[1], so before we try to translate TTDA instructions we will first rewrite all three-input instructions into two-input instructions. Remember that one of the inputs to a three-input instruction must be a constant.

It turns out that there are only three uses of three-input instructions in the TTDA: `form-address-i-store`, `adjust-offset-change-tag`, and two-input instructions with an added **trigger** input. A trigger is a token whose value is ignored—its sole function is to prevent execution of the instruction until the trigger token has been generated.

Figure 4.2 shows the rewriting of `form-address-i-store` and instructions with triggers into two-input instructions. The `form-address` and `i-store` instructions are explained in Section 4.1.6. For now, view the `form-address-i-store` translation as a purely syntactic rewrite. The `gate` instruction is a two-input identity where the right token acts as a trigger. When both tokens have arrived the `gate` forwards the value on the left token to its destination. We deal with `adjust-offset-change-tag` in Section 4.5.

[1] This is not exactly true. Later in this chapter we play a trick to create ETS instructions with more than two inputs. The real restriction is that a matching function cannot attempt to "remember" more than one value at a time, because an activation frame slot can store only a single value.

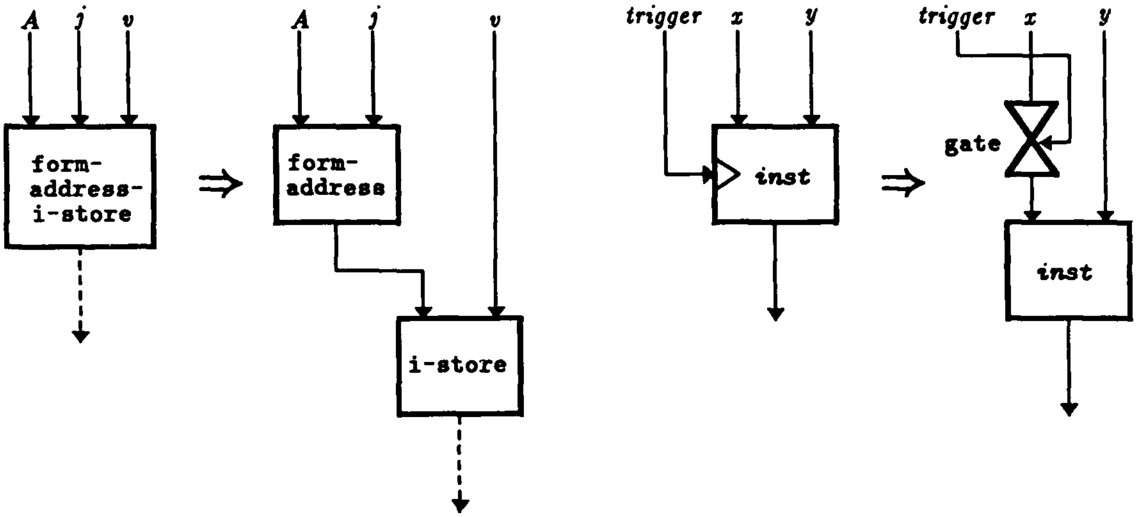


Figure 4.2: Rewriting Three-Input TTDA Instructions into Two-Input Instructions

4.1.4 Operand Matching Rules

In the TTDA, the waiting-matching section is bypassed if either the instruction has a single input or it has two inputs where one is a constant. Notice that the destination instruction does not indicate whether a match should take place because instruction fetch occurs *after* token matching. Instead, TTDA tokens carry an extra one-bit field that indicates the number of tokens required to enable the destination instruction.

Thus, the **source** instructions, the instructions that specify the current instruction as their destination, must indicate the matching rule on the token. This information is encoded on a destination-by-destination basis in the instruction destination list as the number of tokens required to enable the destination instruction, n.

Because the ETS performs instruction fetch before matching, the matching rule W applies to the current instruction. Thus, the TTDA matching rule n specified in the destination lists of the *source* instructions, is folded into the ETS matching rule W specified by the *destination* instruction.

The ETS supports a more general matching function than the TTDA. In fact, once three-input instructions have been rewritten, the entire TTDA instruction set can be implemented with just four basic input operand forms as follows (refer to Figure 4.3):

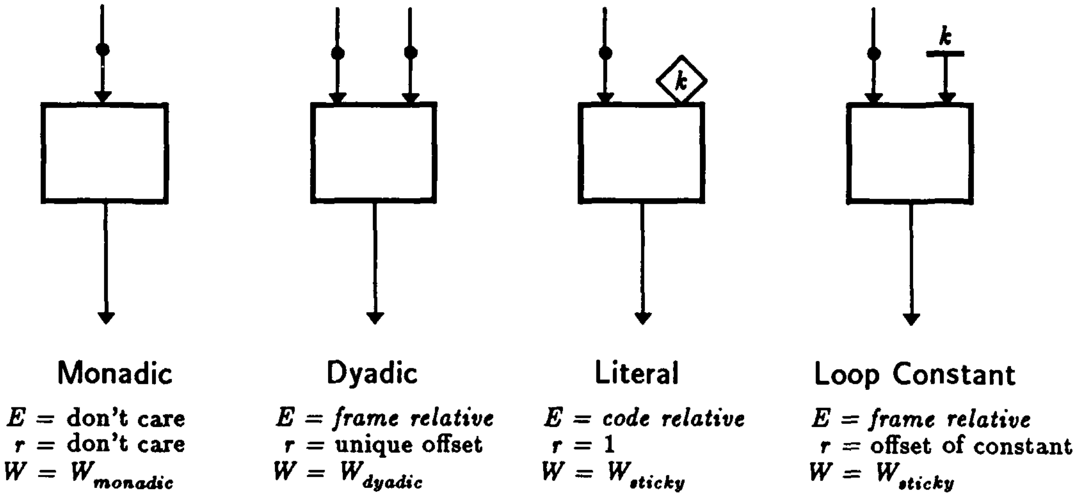
1. **Monadic.** Single-input instructions, ones which bypass the waiting-matching section in the TTDA ($n = 1$) and have no constant input (k irrelevant), use the

$W_{monadic}$ matching rule. The effective address mode, $E.r$, is irrelevant.

2. **Dyadic.** Two-input instructions in which both operands are tokens ($n = 2$), employ the W_{dyadic} matching rule. The effective address mode is *frame relative* where a unique r is assigned to each dyadic instruction in the code block in which the instruction appears (more on slot assignment later).

3. **Literal.** Two-input instructions in which one of the operands is a literal use the W_{sticky} matching rule. The literal value can be placed in the code immediately following the instruction in which case a *code relative* effective address mode is specified ($r = 1$). The match always succeeds because the presence bits on all storage locations which contain instructions are set to *present,* by convention. Alternatively, frequently used literals can be placed at absolute locations and an *absolute* effective address mode used where r is set to the address of the location containing the constant. The constant must be loaded prior to program execution and the associated presence bits set to *present.*

4. **Loop Constant.** Because there is no constant area in an ETS *per se,* loop constants will be compiled as **frame constants**, as explained in Section 4.3. A frame constant is a constant value in the current activation frame, so the effective address mode is *frame relative* with r set to the offset of the constant in the activation frame. The matching rule is W_{sticky}. It is assumed that the match always succeeds, as frame constants are loaded into the activation frame using a `constant-store` instruction prior to the application of arguments to the activation.



Monadic

E = don't care
r = don't care
$W = W_{monadic}$

Dyadic

E = *frame relative*
r = unique offset
$W = W_{dyadic}$

Literal

E = *code relative*
$r = 1$
$W = W_{sticky}$

Loop Constant

E = *frame relative*
r = offset of constant
$W = W_{sticky}$

Figure 4.3: The Four Basic Input Operand Forms

4.1.5 Rewriting Instructions with Three or More Destinations

The **fanout** of an instruction is the number of destination instructions to which it forwards its result, *i.e.*, the length of the destination list. Most TTDA instructions can have an arbitrary fanout whereas an ETS instruction has a maximum fanout of two. The restriction results in fixed-size instructions and permits a single cycle throughput pipeline. Otherwise the number of cycles required to process an instruction would be proportional to the length of its destination list.

Thus, a TTDA instruction with a fanout of greater than two will have to be rewritten as the instruction followed by a **fanout tree** comprising `identity` instructions. Figure 4.4 shows the rewriting of an instruction with five destinations.

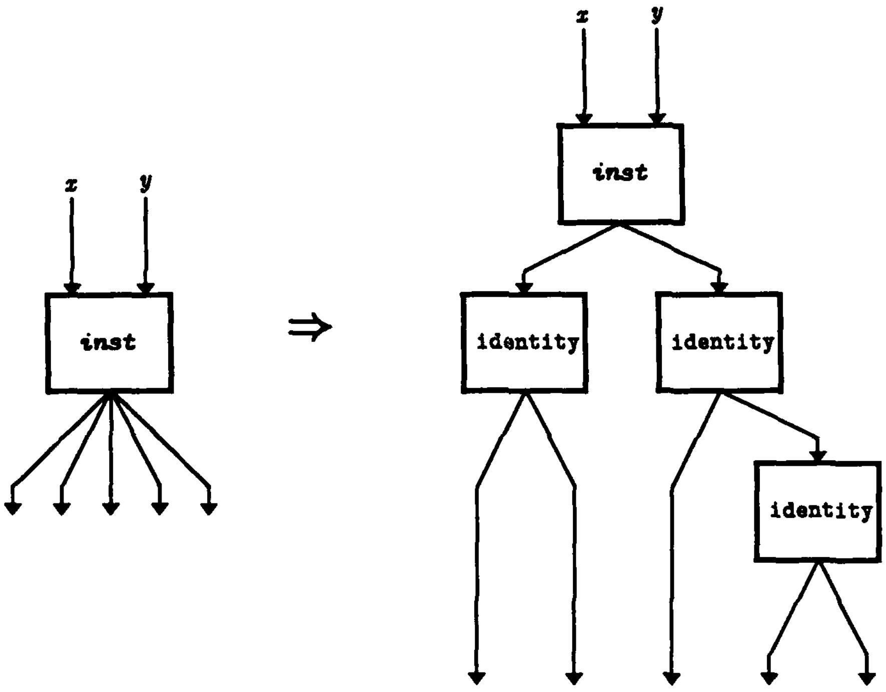


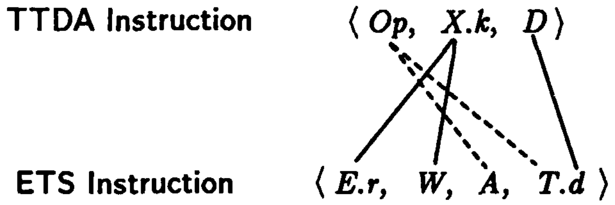
Figure 4.4: A Fanout Tree for an Instruction with Five Destinations

The *average* number of destinations across all instructions executed in a given program cannot exceed two, even when arbitrary fanout is permitted. This is true because the machine graphs are self-cleaning—all the tokens produced by an instruction are eventually consumed—and because an instruction can consume at most two inputs. Thus a program which executes k instructions can produce and consume at most $2k$ tokens, limiting the average number of destinations to $2k/k = 2$.

In practice, restricting fanout to two increases the dynamic instruction count by fifteen or twenty percent. All of the new instructions are, of course, `identities`.

4.1.6 Instruction Opcode Classes

After all three-input instructions have been rewritten and fanout trees have been constructed as needed, we were able to translate the $X.k$ and D instruction components into the $E.r$, W and d ETS instruction fields. These translations are indicated by the solid lines in the following diagram:



The only remaining chore is to translate the TTDA instruction opcode Op. As indicated by the dashed lines, a TTDA opcode will affect both the ALU operation, A, and the token forming rule, T. Most opcodes will employ the normal token forming rule (T_{arith}) that sends the result of an instruction to other instructions within the same context. That is, most opcodes do not directly manipulate the result tag, other than incrementing the statement number (s) and supplying a new port (p). However, an important class of operations, used for intercontext communication, explicitly manipulate the result tag. In this section, we will show how to map most TTDA opcodes into the appropriate ETS A and T fields. We first divide the instruction set into **opcode classes** as follows:

TTDA Instruction Opcode Classes

Class	*Examples*	*ETS Equivalent?*
Dyadic Arithmetic	`add, mult, xor, equal?`	*Yes*
Monadic Arithmetic	`not, neg, fix, float`	*Yes*
Identity	`identity, gate`	*Yes*
Conditional	`switch`	*Yes*
I-Structure	`i-fetch, i-store`	*Yes*
Loop	`d-n, loop-request, d-1`	*No*
Tag Manipulation	`change-tag, form-tag`	*Yes*
Closure	`closure-ready?, closure-chain`	*Yes*
Manager Request	`get-context, make-i-structure`	*Yes*

The serious trouble occurs with TTDA instructions that manipulate a tag's iteration field i (the *Loop* class), as an ETS tag has no separate iteration component. We deal with this problem in the next section where we propose a different loop schema.

Dyadic Arithmetic

A typical dyadic (two-input) arithmetic instruction is integer **add**, which we define as follows:

add
INT × INT ⇒ INT

Inputs:	**Outputs:**
$\langle c.s_l,\ v_l\rangle$	$\langle c.s'_{p'},\ v_l + v_r\rangle$
$\langle c.s_r,\ v_r\rangle$	$[\langle c.s''_{p''},\ v_l + v_r\rangle]$

$A = +$
$T = T_{arith}$

As illustrated above, the **add** instruction expects two integer operands and produces an integer value equal to the sum of the operands. The ALU operation A is $+$, and the token forming rule T is the familiar T_{arith}. Note that for the result tags $c.s'_{p'}$ and $c.s''_{p''}$,

$$\begin{aligned} s' &= s + s1 \\ s'' &= s + s2 \end{aligned}$$

where $s1, s2, p'$ and p'' are encoded by d. The optional second token is generated if a **fanout** of two is required (the number of destinations is encoded by d).

The description suggests that the two input values, v_l and v_r, are provided by input tokens (*i.e.*, a W_{dyadic} matching rule). However, either one could just as well have been a literal or a frame constant. In all of our opcode translation examples, the purpose is to define the ALU operation A and the token forming rule T. The effective address mode E and matching rule W orthogonally compose with A and T, so the same opcode translation works for literals and loop constants.

Another example of a dyadic arithmetic instruction is the **fgeq?** predicate which emits a TRUE value if $v_l \geq v_r$, where v_l and v_r are floating point numbers.

fgeq?
FLT × FLT ⇒ BITS

Inputs:
$\langle c.s_l, v_l \rangle$
$\langle c.s_r, v_r \rangle$

Outputs:
$\langle c.s'_{p'}, v_l \geq v_r \rangle$
$[\langle c.s''_{p''}, v_l \geq v_r \rangle]$

$A = \geq$
$T = T_{arith}$

Monadic Arithmetic

An example of a monadic (one-input) arithmetic instruction is `float`, which converts an integer to a floating point number:

float
INT ⇒ FLT

Input:
$\langle c.s_l, v_l \rangle$

Outputs:
$\langle c.s'_{p'}, float(v_l) \rangle$
$[\langle c.s''_{p''}, float(v_l) \rangle]$

$A = float$
$T = T_{arith}$

Other examples of monadic arithmetic instructions are `fix`, `not`, and `negate`.

Identities

The TTDA has two forms of identity instructions, `identity` and `gate`. The ETS translation of `identity` is a monadic instruction that replicates the input value at its outputs:

`identity`

ANY ⇒ ANY

Input:	Outputs:
$\langle c.s,\ v_l \rangle$	$\langle c.s'_{p'},\ v_l \rangle$
	$[\langle c.s''_{p''},\ v_l \rangle]$

$A = nop$

$T = T_{arith}$

As suggested by the pseudo-type ANY, `identity` works for all machine data types. The `identity` instruction is used for distributing results (*e.g.,* in a fanout tree). The `gate` instruction copies the left operand after waiting for a trigger on the right port:

`gate`

ANY × _ ⇒ ANY

Inputs:	Outputs:
$\langle c.s_l,\ v_l \rangle$	$\langle c.s'_{p'},\ v_l \rangle$
$\langle c.s_r,\ x \rangle$	$[\langle c.s''_{p''},\ v_l \rangle]$

$A = nop$

$T = T_{arith}$

Notice that the trigger and operand can be of any type and that the value of the trigger is irrelevant. We indicate this by a type of "_".

Conditionals

Conditional execution is accomplished by a `switch` operator that has two *mutually exclusive* outputs. A `switch` is a two-input instruction which expects a value and a boolean predicate. The value is copied to the one output if the predicate is TRUE and to the other if the predicate is FALSE[2]:

[2] Contrast this operation with a `compare` followed by a `branch` in many sequential processors. The `compare` side-effects the program status word (*i.e.*, flags) instead of producing a boolean result value. A conditional `branch` would indirectly use the result of the `compare` by examining the program status word.

`switch`
BITS × ANY ⇒ ANY

Inputs:	Outputs:
$\langle c.s_l,\ v_l\rangle$	$\langle c.s'_{p'},\ v_l\rangle$
$\langle c.s_r,\ \mathrm{TRUE}\rangle$	

$A = nop$
$T = T_{switch}$

`switch`
BITS × ANY ⇒ ANY

Inputs:	Outputs:
$\langle c.s_l,\ v_l\rangle$	
$\langle c.s_r,\ \mathrm{FALSE}\rangle$	$\langle c.s''_{p''},\ v_l\rangle$

$A = nop$
$T = T_{switch}$

Notice that the FALSE and TRUE outputs of the `switch` each specify only a *single* destination. Thus, a fanout tree will have to be provided when an output has more than one destination.

Tag Manipulation

There are three basic tag manipulation instructions: `change-tag`, `extract-tag` and `adjust-offset`. The `change-tag` instruction is the primary means for communicating values between contexts:

`change-tag`
TAG × ANY ⇒ ANY

Inputs:	Outputs:
$\langle c.s_l,\ \hat{c}.\hat{s}_{\hat{p}}\rangle$	$\langle \hat{c}.\hat{s}_{\hat{p}},\ v_r\rangle$
$\langle c.s_r,\ v_r\rangle$	$[\langle c.s''_{p''},\ v_r\rangle]$

$A = nop$
$T = T_{send}$

The left operand is of type TAG, $v_l = \hat{c}.\hat{s}_{\hat{p}}$, and the right operand is an arbitrary value. The second token is an optional signal that is emitted in the current context. `Change-tag` and `send` are equivalent instructions. `Extract-tag`[3] is a monadic instruction that generates a token whose value is a tag for the context in which `extract-tag` is executed:

`extract-tag`
$\_ \Rightarrow$ TAG

Input:	Outputs:
$\langle c.s_l,\ x\rangle$	$\langle c.s'_{p'},\ c.s''_{p''}\rangle$

$A = nop$
$T = T_{extract}$

Note that `extract-tag` is restricted to a single output, as the second destination is used as the value part of the first token. The `adjust-offset` instruction can increment the statement number of a TAG type value, so it is really a kind of dyadic arithmetic operation:

`adjust-offset`
TAG × INT $\Rightarrow$ TAG

Inputs:	Outputs:
$\langle c.s_l,\ \hat{c}.\hat{s}_{\hat{p}}\rangle$	$\langle c.s'_{p'},\ \hat{c}.(\hat{s}+j)_{\hat{p}}\rangle$
$\langle c.s_r,\ j\rangle$	$[\langle c.s''_{p''},\ \hat{c}.(\hat{s}+j)_{\hat{p}}\rangle]$

$A = inc\text{-}s$
$T = T_{arith}$

The `adjust-offset-change-tag` instruction is a frequently occurring combination of `adjust-offset` and `change-tag` that is used for passing arguments and returning results. The constant input is invariably the offset adjustment (j, a small positive number) and the instruction can never have more than one output. Thus `adjust-offset-change-tag` can be implemented with the ETS `inc-s-send` (see Equation 3.13 in Chapter 3):

[3] `Extract-tag` is sometimes referred to as `form-tag`.

adjust-offset-change-tag

TAG × ANY ⇒ ANY

Inputs:	Outputs:
$\langle c.s_l,\ \hat{c}.\hat{s}_{\hat{p}}\rangle$	$\langle \hat{c}.(\hat{s} + s1)_{p'},\ v_r\rangle$
$\langle c.s_r,\ v_r\rangle$	$[\langle c.s''_{p''},\ v_r\rangle]$

$A = inc\text{-}s$
$T = T_{send}$

Remember that $s1$ and p' are encoded by the ETS destination $d = s1.p'.s2.p''$. The second token is an optional signal.

I-structure References

There are five I-structure operations (not including allocation): `form-address`, `i-fetch`, `i-store`, `upper-bound` and `lower-bound`. I-structure addresses are represented as tags (refer back to Chapter Three, Section 3.6), so `i-store` is nothing but a `change-tag` with the result port forced to *right:*

i-store

TAG × ANY ⇒ ANY

Inputs:	Outputs:
$\langle c.s_l,\ \hat{c}.\hat{s}_{\hat{p}}\rangle$	$\langle \hat{c}.\hat{s}_r,\ v_r\rangle$
$\langle c.s_r,\ v_r\rangle$	$[\langle c.s''_{p''},\ v_r\rangle]$

$A = nop$
$T = T_{send}$

It is assumed that $\hat{c}$ is the address of the I-structure slot and $\hat{s}$ encodes an I-structure operation (*i.e.*, $\hat{s}$ is the statement number of an I-structure operator). Again, the second token is an optional signal. Similarly, `i-fetch` is a special-case combination of `extract-tag` and `change-tag`:

i-fetch
TAG ⇒ TAG

Input:
$\langle c.s_l,\ \hat{c}.\hat{s}_{\hat{p}} \rangle$

Outputs:
$\langle \hat{c}.\hat{s}_l,\ c.s'_{p'} \rangle$

$A = nop$
$T = T_{extract\text{-}send}$

The tag $c.s'_{p'}$ identifies an instruction in the current context that will ultimately receive the value at location $\hat{c}$. We defer the discussion of `form-address`, `lower-bound` and `upper-bound` until Section 4.3, where we propose an I-structure descriptor convention.

Summary

We have shown the translation of almost every kind of TTDA instruction except those involving loops, I-structure descriptors, closures, and resource managers. We will treat these remaining areas separately, as they require either the adoption of new schemata or a different set conventions. The solutions which follow are not necessarily optimal, the only intent is to give translations that have consistent semantics and reasonable efficiency *vis-à-vis* TTDA.

4.2 Loops

A clear departure of the ETS tag from the TTDA model is the absence of an iteration field. Remember that the TTDA i field (see Chapter Two, Section 2.2) provides a local namespace such that different iterations of the same loop activation can share the same base context c. There are two basic TTDA instructions for manipulating the i field of a token's tag. The `D` instruction sends argument tokens to the next iteration by incrementing their i fields, whereas the `D-reset` instruction returns result tokens to the caller (the **parent context**) by setting i to zero.

A direct way of emulating the effect of `D` is for every iteration to allocate a new context (using `get-context`) for the next iteration and then use a `change-tag` with this context in place of D. Upon termination an iteration issues a **return-context**, returning the context back to the resource manager. If the parent context is passed from iteration to iteration, the final iteration could send the results directly to the parent; so **D-inverse** is also a **change-tag**. As shown in Figure 4.5, a loop compiled this way unfolds as a tail recursion of n activation frames, where n is the number of iterations executed.

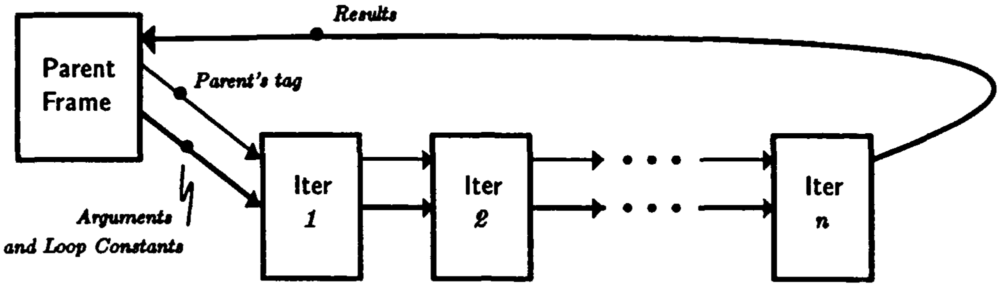


Figure 4.5: Executing a Loop of n Iterations as a Tail Recursion

There are two performance-related drawbacks of this approach: (1) there are n context allocations, and (2) values which are invariant from iteration to iteration (*i.e.,* loop constants—again, refer to Chapter Two, Section 2.2) have to be passed as additional arguments to every iteration.

But there is a way to mitigate these deficiencies by drawing upon a resource-bounded loop compilation technique now employed by the TTDA. Research over the past several years, principally by Culler [18], has clearly demonstrated the benefits of **loop bounding,** the runtime restriction of loop parallelism to at most k concurrent iterations. The bounding value, k, can be set independently for each loop activation and is typically small, say 2 to 20. The greatest benefit of loop bounding is that it can control the resource demands of programs while still revealing enough parallelism to keep a finite collection of processors busy.

Under an ETS, each concurrent iteration of a loop will have a different activation frame. So at any one time k activation frames will be associated with a given invocation of a loop. What we now show is a technique for *recycling* a collection of k activation frames in which the loop constants have been set up as frame constants. Figure 4.6 shows the toplevel strategy.

The preamble (executed in the parent frame) for a k bounded loop allocates k frame contexts, stores the loop constants and parent's tag as frame constants into each of the k

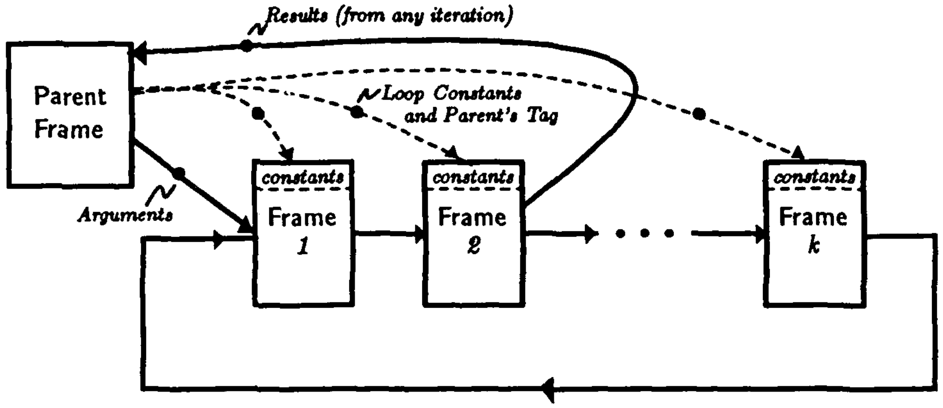


Figure 4.6: Executing a k-Bounded Loop on an ETS

frames, and then links the frames together into a circular queue. The final iteration (any of the k frames) returns its results directly to the parent. The loop postamble deallocates the k frames. This technique is slightly worse than the TTDA in that the setup charge is a function of k rather than the number of processors (the TTDA has to set up loop constants on every processor on which the loop executes). Note that k is always less than or equal to the number of processors on which the loop executes.

There are a lot of details to sort out to make this scheme work correctly. The principal concern is an efficient set of acknowledgment conventions such that arguments are sent to the next frame in the circular queue only when the previous iteration executing in the frame (the current iteration number minus k plus one) has terminated.

The compilation of a single iteration is shown in Figure 4.7. The usual `switch` and `D` operators have been merged into a single `change-tag` for each circulating variable, where the input to the `change-tag` is either the parent context (predicate is FALSE) or the next context (predicate is TRUE). Signals from the `change-tag` instructions are coalesced (by the signal tree) into a trigger which gates a *ready* signal indicating that the iteration has completed and that there are no tokens remaining in the graph (other than the loop constants). In order to correctly process loops with zero iterations (*i.e.*, the predicate is initially FALSE), the 0^{th} iteration bypasses the loop body and inserts its arguments just before the `change-tag` instructions.

The termination signal from each frame is sent to the previous frame as a *ready* signal. Figure 4.8 shows a $k = 3$ bounded loop comprising a set of linked activation frames. The parent context supplies the arguments to the first context in the circular

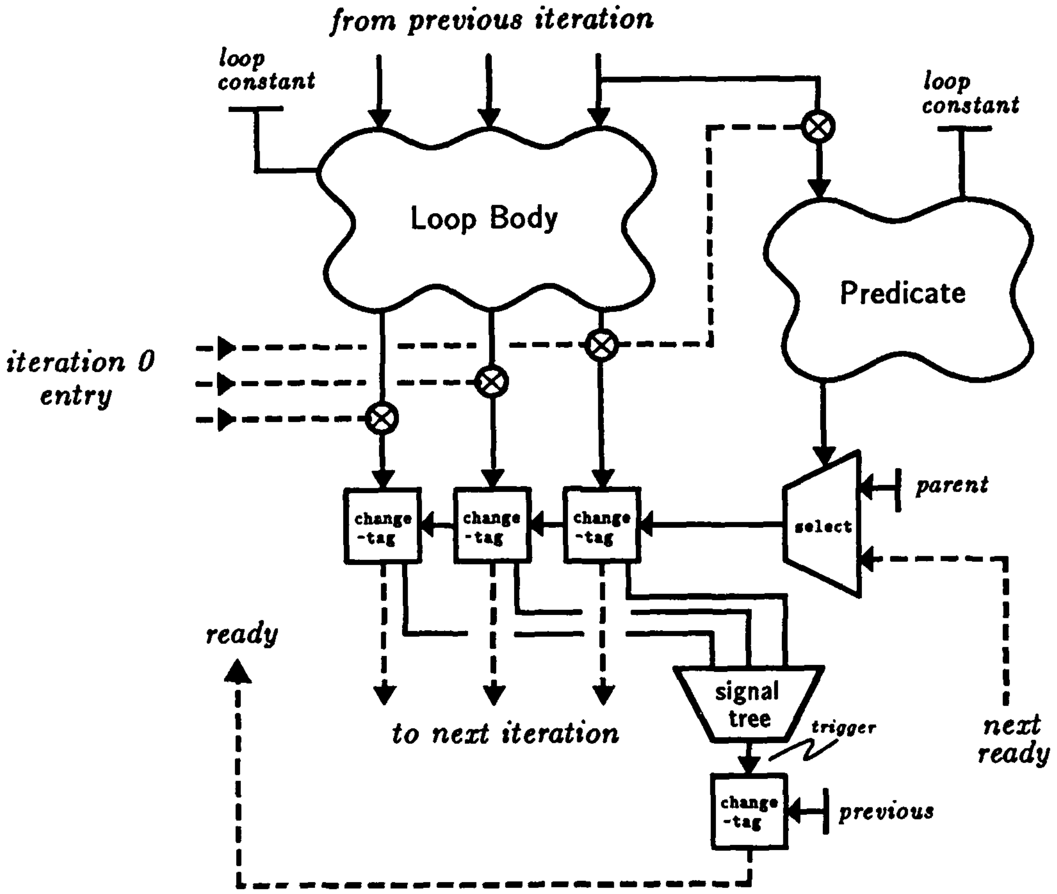


Figure 4.7: Block Diagram of an ETS Loop Schema

list. In general, any context can be the final iteration. This adds complexity to the clean up code wherein the remaining *ready* tokens must be extracted and the the contexts returned to the resource manager. Note that the various loop constants (to the loop body, predicate, *parent*, *etc.*) can be initialized using `i-store` and storing the values directly into the compiler-designated offset in each activation frame.

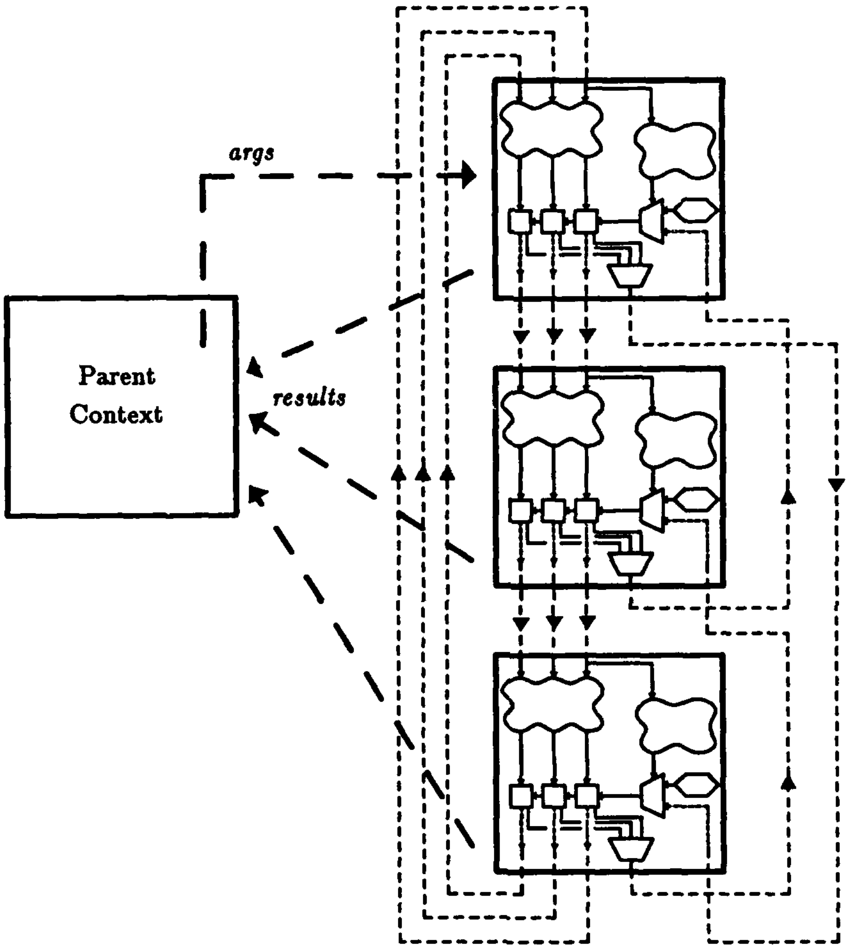


Figure 4.8: A Set of Contexts Forming a $k = 3$ Bounded Loop

4.3 I-structures: Descriptors and Multiple Readers

In section 3.6 of Chapter Three we showed how a **send** instruction can be used to construct an I-structure array where each slot can support at most one deferred reader. Here we give a technique for supporting multiple deferred readers and propose an array header convention that provides the bounds information which was carried on the token under the TTDA model.

4.3.1 Descriptors

The TTDA distinguishes an I-structure **descriptor** from the address of an element in an array. The descriptor contains bounds information as well as a pointer to the base of the array. Bounds checking and lower bound compensation is implicit in the TTDA `i-fetch` and `i-store` instructions.

The ETS model has no separate notion of a descriptor, so a descriptor is an I-structure address (a tag, $\hat{c}.\hat{s}_{\hat{p}}$) which, by convention, points to the **array header.** Figure 4.9 gives an example of a header convention where the lower bound is stored at location $\hat{c}-2$ and the upper bound is stored at $\hat{c}-1$ and the first element in the array is found at location $\hat{c}$.

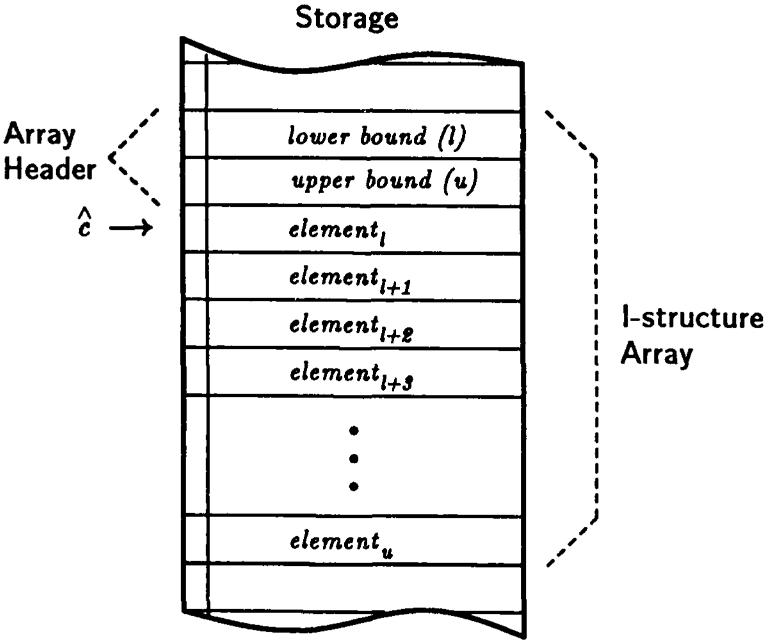


Figure 4.9: Example of an ETS I-Structure Descriptor Convention

Under this convention `form-address`, the instruction which takes a descriptor and an offset and returns an address, is a simple dyadic arithmetic operation on tags (compare with `adjust-offset`, above):

`form-address`

TAG × INT ⇒ TAG

Inputs:	Outputs:
$\langle c.s_l,\ \hat{c}.\hat{s}_{\hat{p}} \rangle$	$\langle c.s'_{p'},\ (\hat{c}+j).\hat{s}_{\hat{p}} \rangle$
$\langle c.s_r,\ j \rangle$	$[\langle c.s''_{p''},\ (\hat{c}+j).\hat{s}_{\hat{p}} \rangle]$

$A = inc\text{-}c$

$T = T_{arith}$

We have glossed over the fact that the TTDA `form-address` incorporates the lower bound in its computation, whereas the above definition only works when the lower bound is zero. Thus, `form-address` will have to be rewritten as shown in Figure 4.10. The lower bound is subtracted from the index yielding a zero-relative offset into the array.

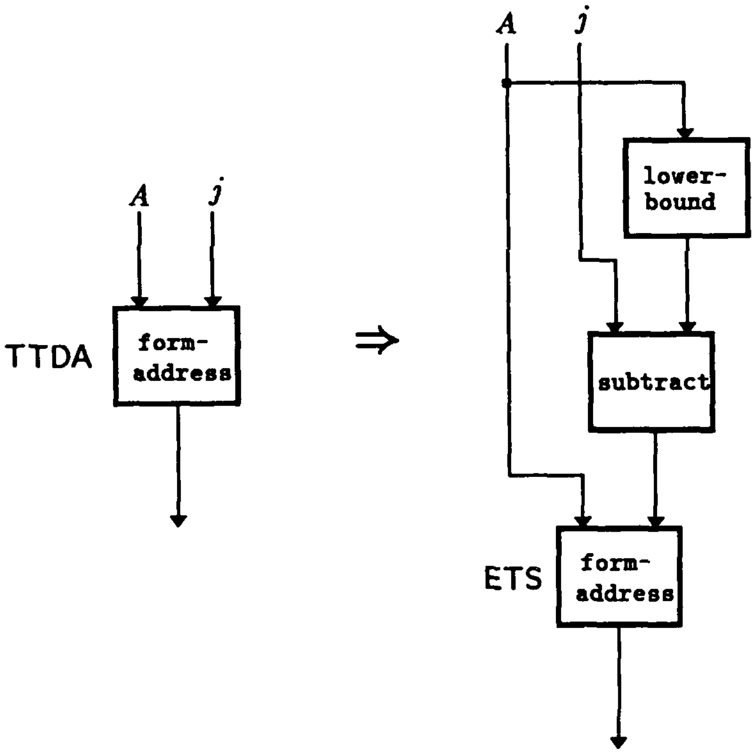


Figure 4.10: Rewriting `form-address` to Account for a Non-Zero Lower Bound

The elision of `form-address` and `i-fetch` to create `form-address-i-fetch` is used extensively:

form-address-i-fetch
TAG × INT ⇒ TAG

Inputs:	Outputs:
$\langle c.s_l,\ \hat{c}.\hat{s}_{\hat{p}} \rangle$	$\langle (\hat{c}+j).\hat{s}_l, c.s'_{p'} \rangle$
$\langle c.s_r,\ j \rangle$	

$A = inc\text{-}c$
$T = T_{send}$

As shown in Figure 4.11 the **upper-bound** and **lower-bound** instructions are just a **form-address-i-fetch** where the offset is the literal constant −1 and −2, respectively.

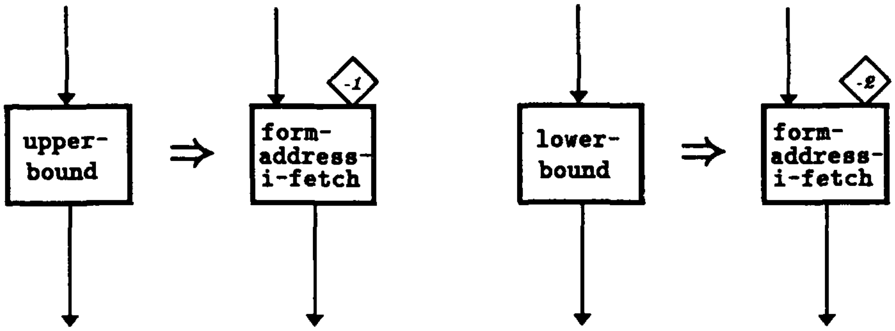


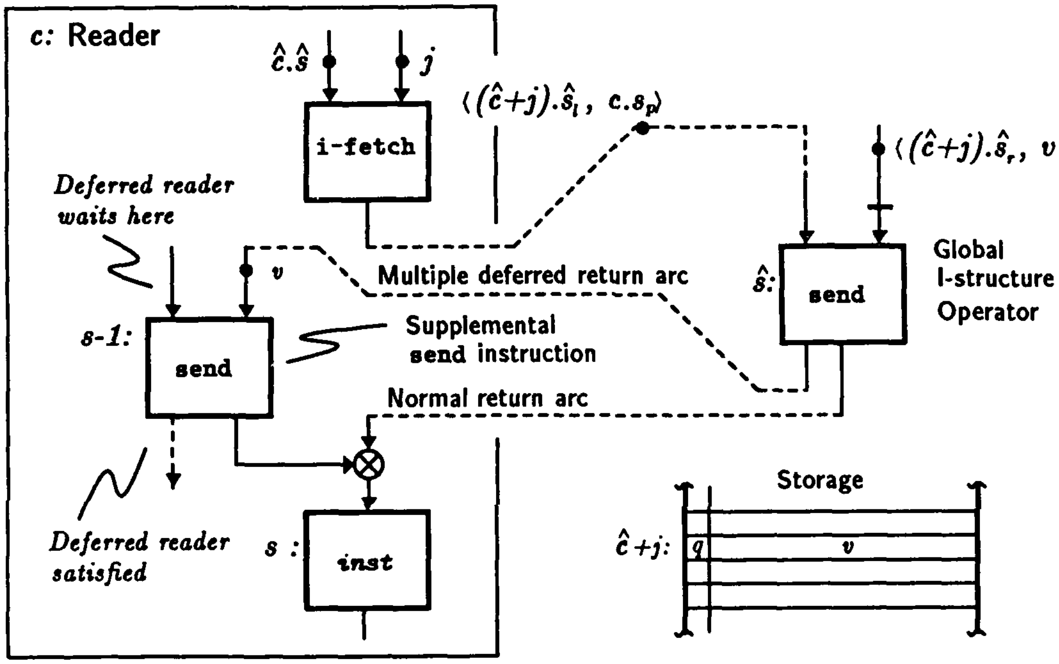
Figure 4.11: Translation of **upper-bound** and **lower-bound**

The TTDA instruction set also specifies that bounds checking should be performed on every **i-fetch** and **i-store**. A naive translation would result in two extra references (one for the upper and one for the lower bound) for *every* fetch or store. While the bounds are likely to be identified as invariants when appearing in a loop, an efficient implementation of runtime bounds checking is a non-trivial compiling problem.

4.3.2 Multiple Deferred Reads

In Section 3.6 of Chapter Three we showed how the actual I-structure *cell* can be implemented as an ETS instruction which is shared across all I-structures. The solution employed a **send** (*i.e.*, **change-tag**) with a W_{sticky} matching rule. A fundamental limitation was the restriction of a single deferred reader of the location, whereas the TTDA calls for the correct handling of several readers of a location before the write takes place.

The TTDA solution is to replace the cell, presently occupied by the first reader, with a pointer to a list of readers. Each subsequent read is pushed onto the list and when the write finally occurs, the list is traversed and each waiting reader is satisfied, and the written value occupies the cell. Not surprisingly, the hardware implementation of an I-structure controller is challenging [25], particularly the automatic management of the deferred read list storage. While it is possible to emulate this behavior on an ETS, we present here a variant that makes the *reader*, not the structure, responsible for the deferred read storage. The basic strategy is to have every reader reserve slots in its activation frame in which a deferred read list can be constructed incrementally.



c: Reader
$\hat{c}.\hat{s}$
j
i-fetch
$\langle(\hat{c}+j).\hat{s}_l,\ c.s_p\rangle$
$\langle(\hat{c}+j).\hat{s}_r,\ v\rangle$
Deferred reader waits here
v
Multiple deferred return arc
$\hat{s}$:
send
Global I-structure Operator
s-1:
send
Supplemental send instruction
Normal return arc
Storage
Deferred reader satisfied
s :
inst
$\hat{c}+j$:
q
v

Figure 4.12: Augmenting an `i-fetch` to Support Multiple Deferred Readers

As shown in Figure 4.12, the instruction preceding every read destination reserves a slot in its activation frame that can record a deferred reader's tag. Now the first read against an unwritten cell causes the usual thing to happen; the reader's return address $c.s_p$ is written into the I-structure slot and the I-structure slot is marked *deferred.* The second reader $c'.s'_{p'}$ sees the *deferred* state and,

1. Decrements the statement number of the second reader's return address and sets the port to *right*, $c'.s'_{p'} \rightarrow c'.(s'-1)_r$,
2. Exchanges the modified tag with the contents of the slot (the first reader's tag, $c.s_p$)

3. Produces the token $\langle c'.(s'-1)_l, c.s_p\rangle$. This token is sent to the second reader's context where it waits on the left port of the supplemental **send** instruction shown in Figure 4.12.

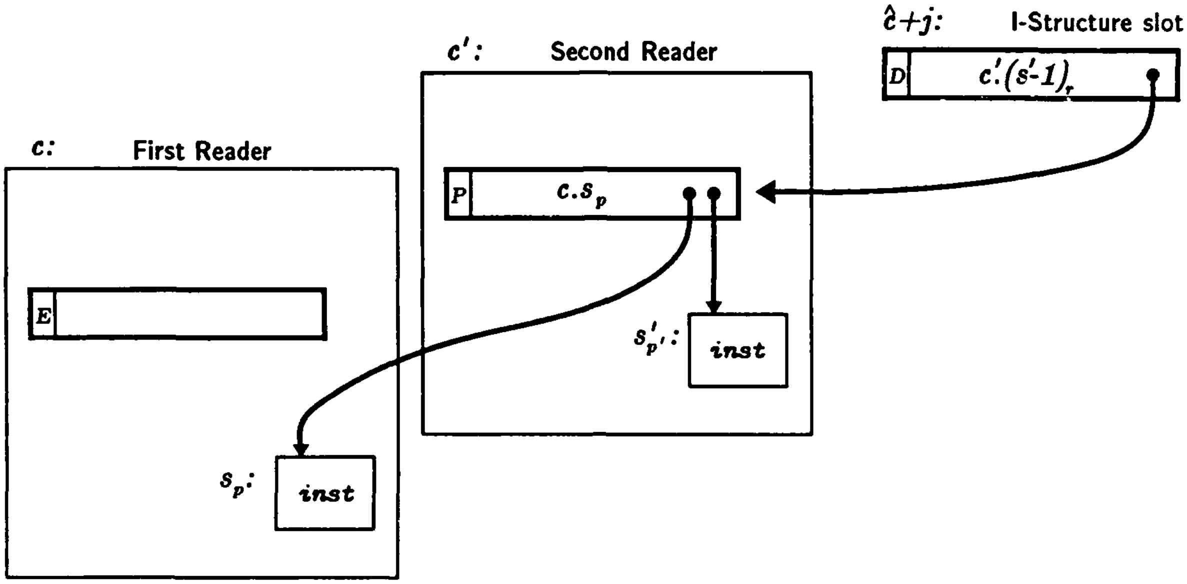


Figure 4.13: A Deferred Read List Comprising Two Readers

Figure 4.13 shows the resulting deferred read list for the two readers. When the write finally arrives to the I-structure slot, the deferred read chain automatically unwinds. The write causes the token $\langle c'.(s'-1)_r, v\rangle$ to be formed (v is the value written), just as if it was satisfying a single deferred reader. This token matches in context c' with $\langle c'.(s'-1)_l, c.s_p\rangle$ and produces two outputs. The first token $\langle c.s_p, v\rangle$ is forwarded on to the next deferred reader on the chain while the second token $\langle c'.s'_{p'}, v\rangle$ satisfies the local reader. The algorithm inductively extends to any number of deferred readers. Figure 4.14 shows the W_{istr} matching function, a variant of W_{sticky}, which supports multiple deferred reads.

4.4 Closures

A TTDA closure is implemented as a chain (a list) of arguments. As arguments are supplied one-by-one, or **curried**, a new chain is constructed whose head is the curried argument and whose tail is original argument chain. A closure is said to be **ready** when the chain has $N-1$ arguments. In this case, the application of an additional argument

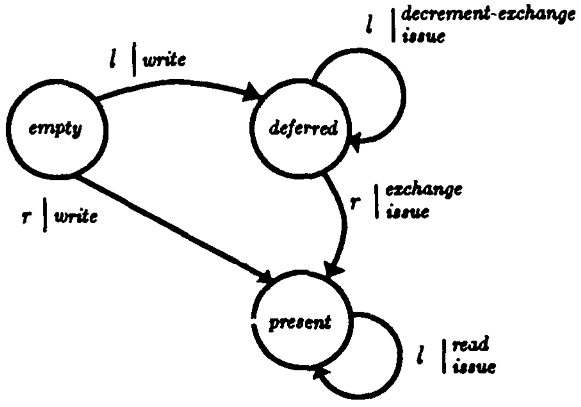


Figure 4.14: The *istr* Matching Function for an I-Structure Slot which Supports Deferred Readers

will satisfy the arity of the function and cause its evaluation. Of course, at compile time it is not always known that a particular application will match the function's arity—so there must be a runtime check to determine if the application of an argument satisfies the arity of the function[4]. Thus the machine representation of a closure value must provide three things: a pointer to the argument chain, a pointer to the function, and an efficient way to determine if the closure is ready. A closure with two arguments applied is shown in Figure 4.15

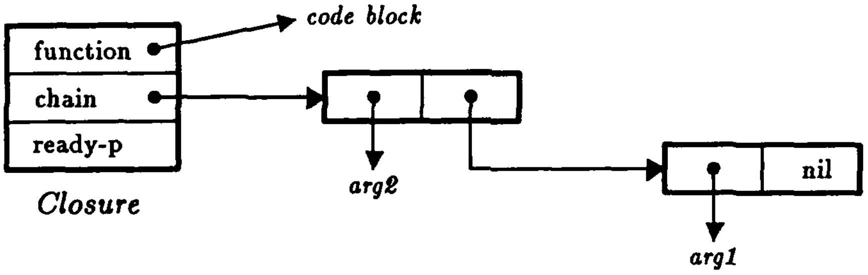


Figure 4.15: A Closure with Two Arguments Applied

Under an ETS a closure is a tag $\hat{c}.s$ where the chain pointer is $\hat{c}$ and the function pointer is s. References to the chain can be made by substituting s with $\hat{s}$ where $\hat{s}$ points to an I-structure operator, as chains are implemented with I-structure cells. The function need only be referenced when the arity is satisfied, in which event a new context is obtained, say c, and a pointer to this activation can be constructed by substituting c

[4] In fact, if arity is known to be satisfied at compile time, the compiler eliminates the closure completely and sends the arguments directly to the activation (the so-called **direct-apply** entry point [49]).

for $\hat{c}$, yielding the tag $c.s$ to function activation. In short,

$$\begin{aligned} closure &= \hat{c}.s \\ chain &= \hat{c}.\hat{s} \\ activation &= c.s \end{aligned}$$

How are we to determine if the closure is ready? There are several possibilities. An initial closure value could have a $\hat{c}$ that points to the arity of the function. To determine whether the closure is ready, the length of the chain is compared to the arity found at the end of the chain. This has the obvious drawback that the `closure-ready?` has an execution time proportional to the number of applied arguments.

Another strategy is to use the least significant bits of s to encode the numbe: of arguments remaining. After each new argument is applied s would be decremented. Then, `closure-ready?` would test the least significant bits of s for zero. This technique would require all functions to have entry points on modulo 2^k instruction boundaries where k is the number of least significant bits of s reserved for the "arguments remaining" counter.

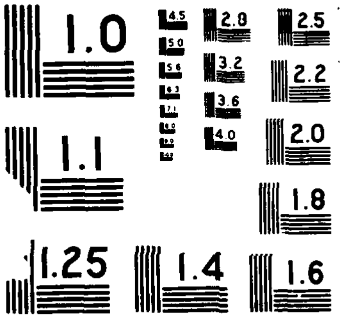
4.5 Resource Managers

The TTDA **get-context** instruction returns a guaranteed-unique context identifier c. Conversely, the **return-context** instruction returns a context identifier to the free poo! of unallocated contexts. Under an ETS, **get-context** and **return-context** respectively allocate and deallocate activation frames. Similarly, the **make-i-structure** instruction allocates a fresh array from I-structure storage. It is essential that dynamic allocation of activation frames and I-structure storage be performed rapidly and efficiently.

These instructions form the **resource manager** opcode class. The TTDA implements resource management instructions as two phase transactions similar to I-structure reads. For example, **get-context** creates a **manager request** token comprising a return address (tag) fcr the requester of the new context. In the TTDA, a manager request is

AD-A207 609   IMPLEMENTATION INDICES (1975-1979)VVOLUME 1(U)   2/2
MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER
SCIENCE G M PAPADOPOULOS DEC 88 MIT/CS/TR-432

UNCLASSIFIED   F/G 12/6   NL


1.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

handled by a "special" (read: "unspecified") processor that consumes the request and (after a while) returns a new context to the requester. Internally the manager is may be non-functional, as it likely maintains mutable data structures representing machine resources.

In an ETS there is no need for a special processor for manager requests. As we show in the following chapter, it is possible to write sequential and/or imperative code—essentially by ignoring presence bits. The key realization is that the manager code will have sections during which it is updating internal state, whereas the manager can be presented simultaneously with an arbitrary number of requests. The problem arises when the manager is processing a request and is in a critical section, but is presented with a new request. The missing functionality is the ability to serialize an *abitrary set* of requests to the manager.

There are many possible solutions to the serialization problem and associated request queueing problem. The first one we present here, based on locks, is appropriate when the manager is very simple (*e.g.*, a free list manager) and it is likely that the queue of requesters never grows very large. Consider the **cs-gate** instruction shown in Figure 4.16. In a normal **gate** instruction, the left token waits until a trigger token arrives on the right port. In the **cs-gate** instruction, however, the instruction fires even in the *absence* of a trigger token, but the result token is sent to an alternate output. This output is looped back to the input to retry the operation.

As a result, the trigger token sequentializes entry into the graph following the **cs-gate**. Concurrent requesters essentially **poll** the entry lock (the trigger token represents the lock). The scheme has an obvious drawback when there are a lot of simultaneous requesters. The requests are dynamically stored in the various processor queues (primarily the token queue) and, of course, an instruction is executed every time the lock is tested.

Another possible solution, appropriate when large queues of requesters may build up or when a request takes a long time to process, is to employ an enqueueing mechanism similar to that used by I-structures that permit multiple deferred readers. A manager request could be a pointer to an I-structure (likely implemented directly in the requesters activation frame) that encodes the identity of the requester, the nature of the request, slots for results *plus* an extra slot in which to build a queue of other requesters. A manager could be triggered on the receipt of such a request, perform the desired operation[5], and

[5] Actually, the manager need not satisfy the request immediately. It can internally defer the request

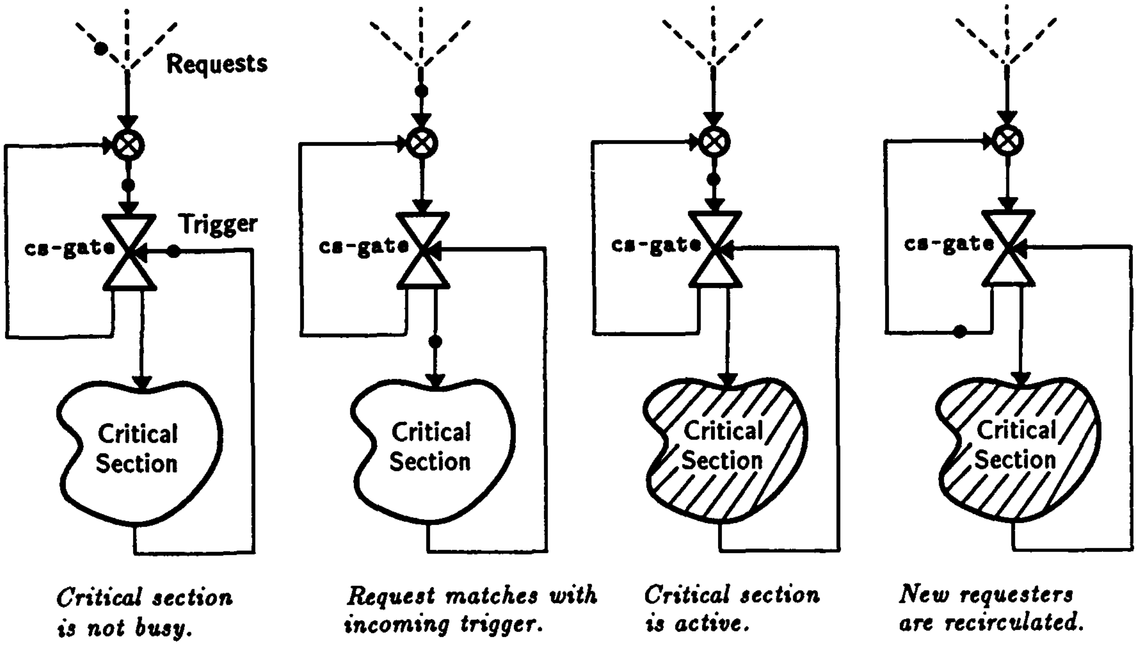

Requests
Trigger
cs-gate
cs-gate
cs-gate
cs-gate
Critical Section
Critical Section
Critical Section
Critical Section
Critical section is not busy.
Request matches with incoming trigger.
Critical section is active.
New requesters are recirculated.

Figure 4.16: Example Operation of the cs-gate Instruction

then indicate its ability to accept another request.

Figure 4.17 illustrates the use a q-gate manager entry point. A trigger token is generated by the manager (the critical section) which always has the value of NIL or "end of list". Suppose that the trigger token is present on the right port of q-gate. The q-gate would then be in *ready* state. A request (of the form of a pointer to a request I-structure) is presented to the left port. This causes the q-gate state to change to *busy* and the request to be processed. Notice that the location associated with the q-gate is still storing the NIL value from the trigger. Now if the request is processed and a new trigger is emitted before any other requests are made, then the q-gate state changes back to *ready* upon receipt of the trigger (and no new token is generated for the manager).

If, however, another request arrives while in the *busy* state then this request is *queued* just like an I-structure deferred reader (see Figure 4.13) except that the last element in the list is now NIL. Subsequent requests are similarly enqueued. When the trigger is finally emitted by the manager the trigger (NIL) is exchanged with the head of the list presently occupying the q-gate slot, the state is set to *busy*, and a token is generated for the manager. This token is a pointer to a *list* of requesters and is sent to a different entry

and then indicate its ability to accept a new request. In fact, we *expect* that most sophisticated resource managers will do this.

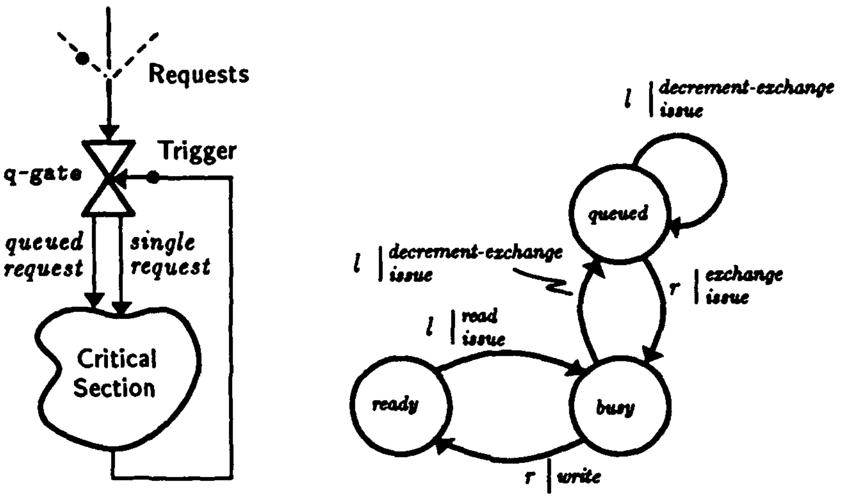


Figure 4.17: Enqueueing Manager Requests with **q-gate**

point in the manager. The manager walks down the list processing requests one-by-one until NIL is reached. When all requests have been processed, a trigger is emitted again to the **q-gate**. Notice that if more requests had enqueued in the meantime, then the manager would be activated again with another list. Otherwise, the **q-gate** would return to *ready*.

4.6 Operators with More than Two Inputs

So far we have restricted our attention to direct translation of TTDA instructions without taking advantage of the ETS. In this section we try to give the flavor of the kinds of optimizations that can be performed. In particular, we investigate the construction of ETS instructions with more than two inputs and show how to create a **gate** that requires *three* trigger tokens to arrive before firing.

While it is sometimes possible to construct operators with more than two inputs, the associated firing rules must account for the fact that a location can store only a single value. For example, the **gate** instruction forwards the value on the left port when a token is present on the right, *independent* of the value of the right token. More simply, the only role of the right token is to cause a side-effect on the presence state. It is easy to design a matching rule that requires more than one right token before the instruction fires. It *counts* the total number of tokens and fires after the fourth token arrives, issuing an activity comprising a value equal to that of the left token. The instruction schematic and wait-match state machine are given in Figure 4.18.

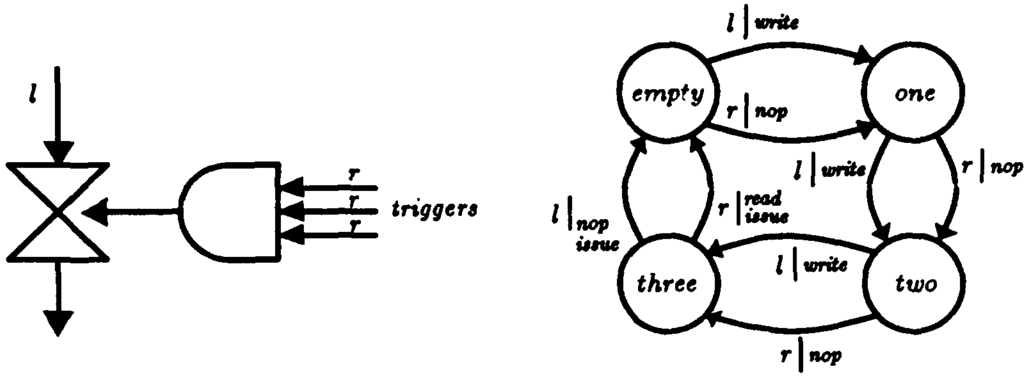


Figure 4.18: A **gate** Instruction With Three Triggers

In the above example, all trigger tokens were presented on the right input. A tag can only encode an l or r port, but suppose we wanted to build an instruction with a third distinguished port? Again following the restriction that a matching function can't try to remember more than one value, we can have consecutive instructions with *different* wait-match functions share the *same* location (*i.e.*, specify the same r), and thus communicate through a common set of presence bits. For example, we can contrive a new gate that normally fires only when a trigger is present but can be *short circuited* into firing independent of a trigger. (Refer to Figure 4.19)

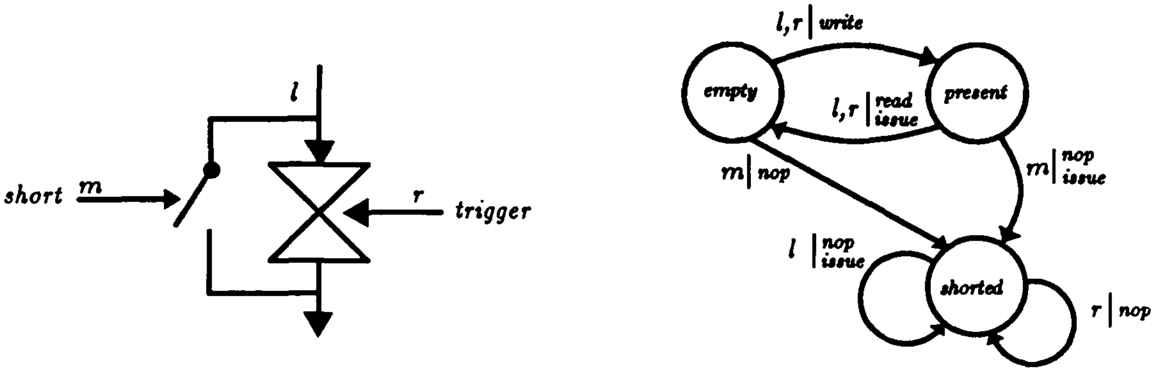


Figure 4.19: A **gate** Instruction which can be Short-Circuited

This is implemented with two instructions, a modified **gate** instruction that is sensitive to the *shorted* state but is unable to enter or leave the *shorted* state[6], and a **toggle**

[6] Thus all **gate** instructions can be implemented this way and only those that have a short circuit option will be followed by the **toggle** instruction.

instruction that can place the **gate** into the *shorted* state (refer to Figure 4.20). The composite instruction is formed by having the **gate** and `toggle` instruction share the same presence bits, *i.e.*, they specify the same r.

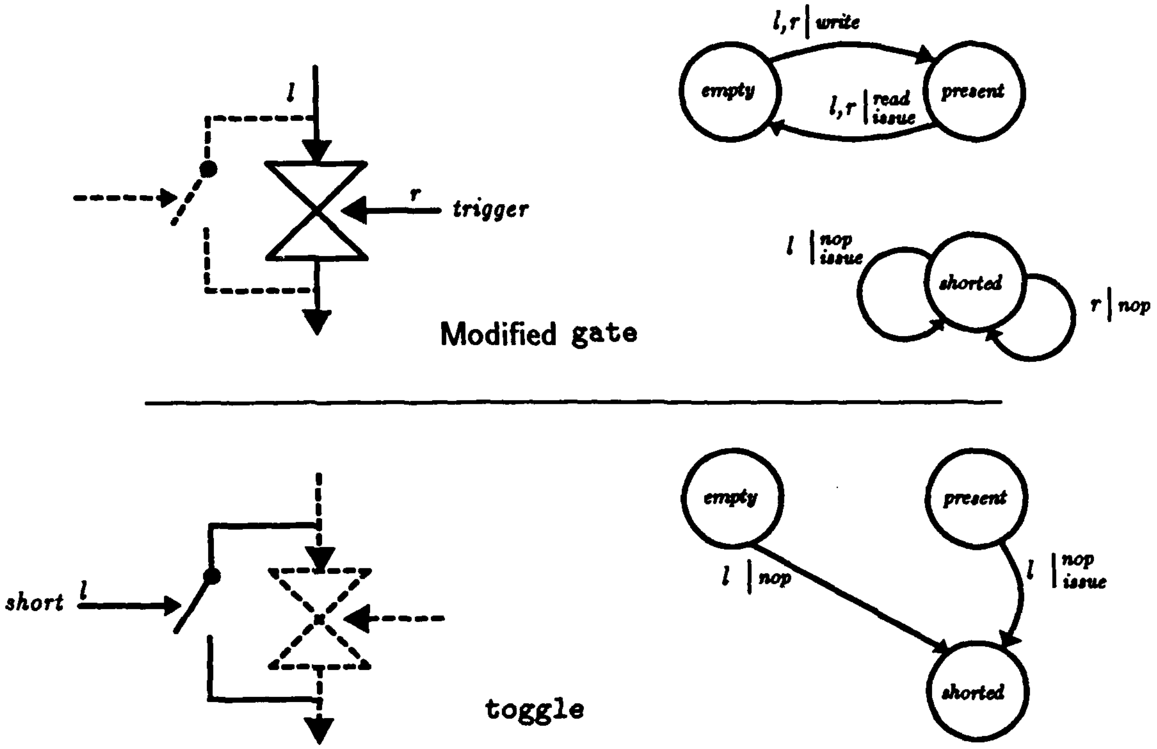


Figure 4.20: Composition of Two Instructions to Make Short-Circuited **gate**

4.7 Slot Allocation Revisited

The generation of machine code for a code block is straightforward. The only real issue is the assignment of a frame offset r for each dyadic instruction in the code block. A trivial way to assign frame offsets is to give each dyadic instruction (those with non-constant inputs) a unique r. In this section we show how to reduce the size of an activation frame by assigning, where possible, the same r to different instructions in the code block.

In section 3.2 we defined a code block as a digraph $G = (E, V, I, O)$, where $v_j \in V$ is an instruction with a maximum indegree and outdegree of two, and $(v_j, v_k) \in E$ implies $v_j \rightarrow v_k$. I and O are the set of input and output arcs, respectively. G is also required to be self-cleaning and acyclic, or at least no valid evaluation order of G should result in two tokens occupying the same arc.

The normal ETS firing rule for dyadic operators is self-cleaning—after a match takes place the presence state is reset to *empty* and is thus able to perform a new matching cycle. To minimize the size of on activation frame we would like to reuse a given location several times during the execution of a code block. That is, we would like to assign the same r to different instructions while still preserving correct operation over *all* possible execution orders. For two instructions, v_j and v_k, to share the same r within an activation, it must be guaranteed that if a token is present on an input to v_j then *no* token can possibly be present on the input of v_k. There are two ways that such mutual exclusion can arise,

- There is a **double dependence** between v_j and v_k. That is, *both* inputs of v_k are dependent, through some path, upon the output of v_j.
- Instructions v_j and v_k appear on opposite sides of a conditional.

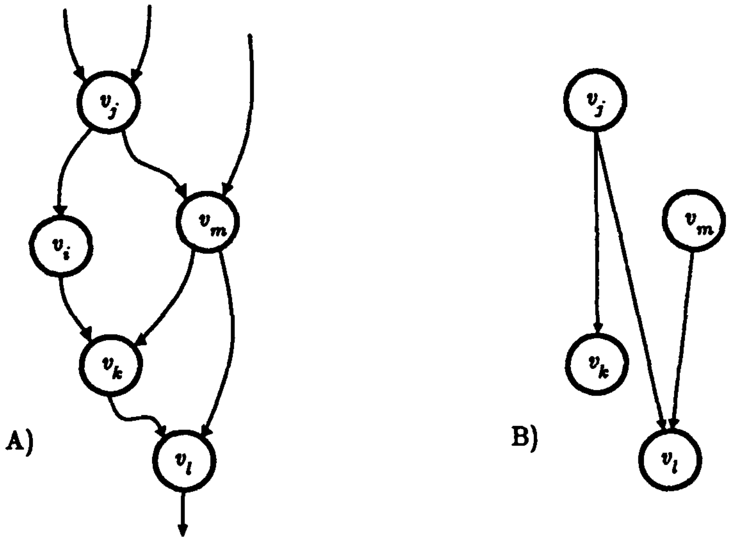


Figure 4.21: A) Code Block and B) Corresponding Double Dependence Graph

Figure 4.21 shows a simple code block and its corresponding double dependence graph. Notice that both v_k and v_l are double dependent upon v_j but there is no double dependence between v_k and v_l. This would permit v_k **or** v_l to be assigned the same r as v_j, but then the remaining instruction (v_l or v_k) would have to be assigned a *different* r. The optimal assignment problem is readily apparent. If we were to assign v_j and v_k the same r then we could also allow v_m and v_l the share a frame slot, so this code block requires an activation frame with two locations. However, if we assigned v_j and v_l the same r then, because there is no double dependence relating v_m and v_k, the code block would require and activation frame with *three* locations.

Plotkin [43] shows that the optimal assignment without conditionals is reducible to the maximum matching problem for a bipartite graph, which is solvable in polynomial time. Starting with the original graph, $G = (V, E, I, O)$, we create the double dependence graph $G' = (V, E')$ such that $(v_j, v_k) \in E'$ when a double dependence exists from v_j to v_k. G' can be transformed into a bipartite graph $G'' = (V, U, E'')$ where,

$$\begin{aligned} U &= V \\ E'' &= \{(v_j, u_k) | (v_j, v_k) \in E'\} \end{aligned}$$

Determine M as the maximum matching for G''. Now M has the minimum number of connected components of G'', and all instructions within a connected component may be assigned the same r. Plotkin also shows that the maximum number of slots required is no worse than the max cut of the original dataflow graph. This means that the storage required by an ETS activation frame is no worse than *maximum* amount of storage required by a fully associative token store. The *average* or instantaneous amount of storage required by a fully associative token store may be substantially less than an ETS, but we note that ETS activation frame slot can be constructed from standard random-access memory which is considerably less expensive (in terms of circuit size) than associative memory.

4.8 Summary

To convert dataflow graphs for execution on an ETS, the following changes should be made to the Id-to-TTDA compiler [49]. These changes can be made (except 5) while still producing correct TTDA code:

1. The peephole optimizations that generate three-input instructions should be disabled.
2. The instruction fanout module should be modified to generate fanout trees as required.
3. All structures should be compiled using an array header convention and all lower bounds adjustments and bounds checking should be explicit.

4. Change the bounded loop schema to perform activation frame recycling similar to what we have proposed.
5. Add a compiler module to assign frame offsets.

In summary, we have shown a complete, albeit brute-force, way of converting TTDA machine graphs into ETS machine graphs. There is clearly a lot of room for improvement, but we believe that our basic objective has been met—the ETS has no fundamental loss of expressiveness compared with the TTDA. The goal of the next chapter is to show, in fact, that the ETS is a *superset* of the TTDA.

Chapter 5

Compiling Imperative Languages for an ETS

An ETS processor is more general than a TTDA graph interpreter in that there is contained local state, the activation frame, on which imperative operations may be performed. In this chapter we reformulate the ETS as a multithreaded sequential processor in which context switching, forking and joining are extremely efficient. There are both pedagogical as well as practical reasons for developing this view. Comparing the machine to more familiar sequential engines provides deeper understanding of the strengths, and more importantly, the inherent weaknesses of the ETS. From a practical standpoint, a familiar imperative instruction set can simplify the coding of resource managers.

5.1 Threads

An ETS token, $\langle c.s_p, v \rangle$, can be viewed as a **thread descriptor** where,

s The program counter.

c The frame pointer.

v The "accumulator."

As shown in Figure 5.1, a thread descriptor comprises two words. The frame pointer c points to an activation frame containing a set of registers, $r0$, $r1$, $\ldots$, and the program counter s points to the next instruction to execute.

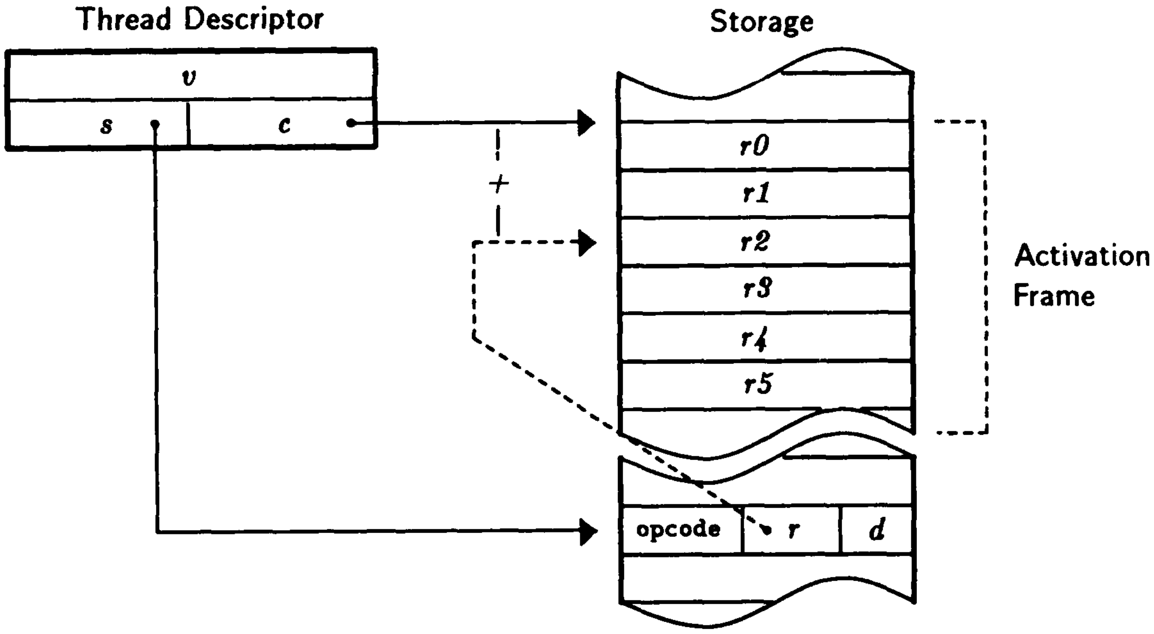


Figure 5.1: Viewing an ETS Token as a Sequential Thread Descriptor

The value v can be thought of as an accumulator in the following sense. If the imperative $W_{write!}$ matching function is specified (refer back to Chapter Three, Section 3.4) then v, the accumulator, is written into register r (from the instruction). If a W_{read} matching function is specified then register r is read and its contents and the accumulator can be processed by the ALU, yielding a new value for the accumulator. The matching functions *always* issue an activity. If all instructions have only a single destination then the program executes as a serial stream of operations where an instruction from the stream can specify,

1. A single operation against the activation frame: a write of the accumulator into a frame register, the read of a frame register, or an atomic exchange of the accumulator with a frame register.
2. An operation on the accumulator and the value read from the frame register to yield a new accumulator.
3. An increment to the program counter.

We will describe thread programs using an assembly language instead of dataflow graphs (the graphs will be a straight sequence of one-input, one-output operators). The `STORE r` instruction causes the accumulator to be written into frame register `r` and does not change the value of the accumulator:

`STORE r`
ANY ⇒ ANY

Input:	Output:
$\langle c.s,\ v\rangle$	$\langle c.(s+1),\ v\rangle$

$E =$ *frame relative*
$W = W_{write!}$
$A = nop$
$T = T_{arith}$

The `LOAD r` instruction loads the accumulator with the contents of register `r`:

`LOAD r`
_ ⇒ ANY

Input:	Output:
$\langle c.s,\ x\rangle$	$\langle c.(s+1),\ (r)\rangle$

$E =$ *frame relative*
$W = W_{read}$
$A = nop$
$T = T_{arith}$

The expression (r) should be read as "the contents of register r". Recall that W_{read} does *not* reset the presence state to *empty,* so a register can be read any number of times. `LOAD r` is a special case `OP r`, which performs an operation on the accumulator and the contents of register `r` and places the result in the accumulator. For example, `ADD r` adds register `r` to the accumulator:

`ADD r`
INT × INT ⇒ INT

Input:	Output:
$\langle c.s,\ v\rangle$	$\langle c.(s+1),\ v+(r)\rangle$

$E =$ *frame relative*
$W = W_{read}$
$A = +$
$T = T_{arith}$

By changing the effective address mode, we can generate a family of instructions that operate on the accumulator using constant (literal) values. For example, LOAD *const* is just like LOAD r, except for using an instruction relative effective address mode (assuming *const* appears in the code at $s + r$):

LOAD *const*
$\_ \Rightarrow$ ANY

Input:	Output:
$\langle c.s,\ x\rangle$	$\langle c.(s+1),\ const\rangle$

$E = instruction\ relative$
$W = W_{read}$
$A = nop$
$T = T_{arith}$

In the following section we develop a complete imperative instruction set by showing how to translate a common intermediate compiler form into ETS instruction sequences.

5.2 Translating Quads into Thread Sequences

Quads are an intermediate description language frequently used by a compiler as an abstract machine language [1]. The code generation phase of a compiler translates quads into the target machine instruction set. In this section we show how to translate quads into ETS instruction sequences. We will introduce new instructions as we need them.

Definition 5.1 *A* **quad** *is a tuple,* $\langle Op, res, a_1, a_2\rangle$*, where,*

Op Is an operation like `*, +, :=, fetch, store, branch,` *etc.*

res The result, either a register or *null.*

a_1 The first argument, always a register.

a_2 The second argument, either a register, a label, a literal or *null.*

A thread comprises a sequence of quads that are to be executed in order unless there is a **branch**. We will assume that there are enough slots in an activation frame to name all the registers. We now give the translations for four different classes of quads:

1. **Simple Assignment.** Copying a value from one register to another register, $r_i \leftarrow r_j$
2. **Arithmetic.** Three address arithmetic operations, $r_k \leftarrow r_i \; Op \; r_j$.
3. **Indirect Assignment.** Loading (or storing) a register with the contents of a location whose address is specified by another register, $r_i \leftarrow [r_j]$ and $[r_i] \leftarrow r_j$.
4. **Conditional Branch.** Conditionally setting the program counter (s) in response to a test of the contents of a register.

Simple Assignment

A **simple assignment** is given by the quad $\langle$:=, r_j, r_i, *null*$\rangle$ which copies the contents of register r_i into r_j. This takes two ETS instructions, a LOAD followed by a STORE:

```
LOAD   ri                   ;; v <- reg i
STORE  rj                   ;; reg j <- v
```

The LOAD sets the accumulator to the contents of r_i and the STORE writes the value of the accumulator into register r_j.

Arithmetic Quads

The **arithmetic** quads are "three address" operations $\langle$Op, r_k, r_i, $r_j\rangle$ which set register r_k to the value obtained by performing the indicated arithmetic operation Op on the contents of r_i and r_j. The direct transliteration into a "one address" accumulator instruction set takes three instructions,

```
LOAD   ri                   ;; v <- reg i
OP     rj                   ;; v <- v OP reg j
STORE  rk                   ;; reg k <- v
```

Better code can be generated when the result of the operation, r_k, is one of the arguments of the next quad. If r_k is just a temporary (its value is only needed by the next quad), then the STORE of the first operation and the LOAD of the second could be eliminated. That is, the following sequence of arithmetic quads where r_k is a temporary,

$$\langle \texttt{Op1},\ r_k,\ r_i,\ r_j\rangle$$
$$\langle \texttt{Op2},\ r_m,\ r_n,\ r_k\rangle$$

can be translated into four ETS instructions:

```
LOAD   ri                ;; v <- reg i
OP1    rj                ;; v <- v OP1 reg j
OP2    rn                ;; v <- v OP2 reg n
STORE  rm                ;; reg k <- v
```

Indirect Assignment

Indirect assignment has two forms. An **indirect load** $\langle :=, r_j, [r_i], null\rangle$ copies the contents of the location whose address is in r_i into r_j. An **indirect store** $\langle :=, null, [r_i], r_j\rangle$ copies the contents of r_j into the location whose address is in r_i. It is assumed that a value which is used as an address is of type TAG.

The two phase memory transactions still play an important role in the sequential thread model—except now we ignore the I-structure semantics and imperatively read and write storage. An indirect store expands into two ETS instructions:

```
LOAD   rj                ;; v <- reg j
STORE [ri]               ;; [reg i] <- v
```

The STORE [r] instruction is just a **change-tag** with a signal output:

STORE [r]
ANY × TAG ⇒ ANY

Input:	Output:
$\langle c.s,\ v\rangle$	$\langle (r)_r,\ v\rangle$ $\langle c.(s+1),\ v\rangle$

$E = \textit{frame relative}$
$W = W_{read}$
$A = \textit{nop}$
$T = T_{send}$

STORE [r] has two outputs. The first token $\langle (r)_r,\ v\rangle$, when processed, causes the location $\hat{c}$ to be written with the value of the accumulator v. It is assumed that $(r) = \hat{c}.\hat{s}$ where $\hat{s}$ points to an I-structure operator that permits multiple writes (the value is sent to the right port). The second token, the signal, is the continuation of the thread. There is a subtle point concerning this signal. The TTDA specifies the signal to be generated when the instruction executes rather than when the store actually takes place. In an imperative model, it is important to preserve the *order* of operations on memory—a property not guaranteed by TTDA semantics. An implementation may choose, however, (Monsoon does, see Chapter Six) to guarantee that memory operations will always take place in the order generated, thus permitting the safe execution of imperative threads.

To read a storage location, we form a read message (a token) with our **continuation,** the tag $c.(s+1)$. The location is read and our thread is continued with the accumulator equal to the contents of the location:

```
LOAD    [ri]                ;; v <- [reg i]
STORE    rj                 ;; reg j <- v
```

The LOAD [r] instruction is just like i-fetch:

LOAD [r]
_ ⇒ ANY

Input: $\langle c.s,\ x\rangle$

Output: $\langle (r)_l,\ c.(s+1)\rangle$

$E =$ *frame relative*
$W = W_{read}$
$A = nop$
$T = T_{send}$

By analogy, we can employ `form-address-i-fetch` to specify index plus offset addressing,

```
LOAD   ri                  ;; v <- index
LOAD   [rj + v]            ;; v <- [rj + index]
STORE  rk                  ;; reg k <- v
```

Conditional Branches

A conditional branch is $\langle$**branch**, *null*, *label*, $r_i\rangle$ which begins executing the quad at *label* if the contents of r_i are TRUE. This translates to a single ETS instruction when the label is known at compile time:

```
BRANCH  offset, ri  ;; s <- s + offset, if reg i = TRUE
```

BRANCH *offset*, `r` is no more than a `switch` instruction where the predicate is in register `r`. Unconditional branches are unnecessary as every instruction can specify an arbitrary program counter increment. A label that is computed at run time is represented as a tag $c.s'$. Control can be transferred to the label using `change-tag`.

5.3 Multiple Threads Within an Activation

Many threads can be simultaneously executing out of the same activation frame. Each thread will have its own program counter and accumulator but can share information with other threads through registers in the activation frame.

A thread can be **forked** by specifying two destinations. In this event, the frame pointer and accumulator are copied and a new thread is created with a differing program counter.

Two threads can be **joined** by selecting a rendezvous register within the activation frame, specifying a W_{dyadic} matching function, and the *same* program counter, *i.e.,* a normal two-input dataflow operator. In a join, the first arriving thread deposits its accumulator into the register and "dies". The second arriving thread reads the register and continues execution, able to perform an operation on its accumulator and that of the other joined thread. So a single instruction can specify a join of two threads, an operation on the accumulators, and a fork of two threads.

Because the indirect load is a two phase transaction, it is also possible to fork the LOAD in the same instruction—the thread continues computing while the load is taking place. By using the presence bits on the register being loaded, the thread can automatically resynchronize to the load. It is likely that a compiler will find the most opportunities for introducing threads around loads. The model is really quite powerful here as it allows arbitrary "delayed load" of registers from memory, and the processor never blocks waiting for an interlock.

Communication between activations is accomplished by the change-tag instruction which *creates* a thread that is automatically associated with the target activation. Linkages between activations can be maintained by passing tags as continuations.

5.4 Summary

Many of the quads translate into more than one ETS instruction. In contrast, there is nearly a one-to-one relationship between quads and the instruction set of a conventional three address von Neumann processor. The extra ETS instructions are due to the restricted addressing—only a single register can be read or written in a cycle whereas a three address machine can read two registers and write one register.

A surprising result is that the dynamic instruction counts for a program compiled to a dataflow graph are almost equivalent to the same algorithm compiled onto a three address von Neumann processor [8]. How can this be when we would clearly executc more instructions if the sequential transliterations given in this chapter are used?

When an expression is decomposed into a maximum number of threads, *i.e.*, a dataflow graph, then the tokens carried on arcs themselves form a large set of "temporaries": *lots* of accumulators[1]. This storage is not free—it is present in the processor as the token queue. Individual dataflow instructions look more like three address operations: left operand plus right operand yields new operand. When we naively translate sequential code we take no advantage of this extra storage.

The imperative translation is a nonetheless useful one as it provides an expedient path for coding resource managers. It also gives some insight in how we might improve the ETS by permitting more complex (*i.e.*, three-address) frame operations [41]. Even without thesr extensions, it is clear the ability to execute imperative code makes the ETS a strict superset of the TTDA. It is likely that an ETS extended to three address arithmetic may finally close the gap between dataflow machines and the emerging dataflow/von Neumann hybrid architectures [28].

[1] What we lose on each instruction, we make up in volume!

Chapter 6

Monsoon: An ETS Multiprocessor

The ETS is more of a computational model than a description of a multiprocessor. For example, the ETS does not specify the size of words (only that machine data types are the same fixed size) nor does it even suggest how execution should be partitioned across multiple processing elements. Indeed, there are a host of "bindings" that have to be made when designing a machine based on ETS principles:

- How can the machine be partitioned into separate processors? Is code shared or replicated?
- How are tags to be represented? How big do they need to be? What is an appropriate word size?
- How much memory does a processor require? What are the relative sizes of code, activation frame and token queue storage? Can the memory be cached?
- What kind of exceptions should the hardware detect? How can exceptions be handled? Should the machine support hardware type-tags?
- What are the requirements for the interprocessor network? What are the bandwidth demands *vis-à-vis* the processor performance? Are there any ordering constraints on messages?
- What kind of instrumentation should be provided?

Finding good answers to these design questions is often more of an art than a science. In the best of worlds, we would perform detailed simulations of our multiprocessor in order to analyze implementation alternatives. However, a basic dilemma emerges. The performance of a uniprocessor can be measured against a standard set of instruction traces that represent the spectrum of applications over which the processor is expected to execute. Simulating a uniprocessor over a few million instructions would instill a good deal of confidence in the machine's performance for a given setting of design parameters.

For a multiprocessor a few million instructions would barely scratch the surface, because the *interactions* that we want to observe occur at much coarser grain than instructions. In the multiprocessor setting we need to evaluate the evolution of processor state, the frequency and type of interactions with other processors, the latency, effective bandwidth and potential for congestion in the network, *et cetera*. The computational demands of such a simulation are overwhelming, requiring a very high performance (read: parallel) simulation engine. In fact, it appears that the best simulation for a parallel processor is the processor itself!

The **Monsoon** processor is a first attempt at a reasonable *prototype,* a machine which should provide an environment for collecting data and analyzing design tradeoffs. As such, we have valued flexibility over optimizing performance or cost. Conversely, we have to be "honest" about the practicality of the design—it has to be clearly *possible* to develop a cost/performance competitive machine. Thus, we favor a true multistage pipeline design versus a microcoded implementation. A microcoded machine would be easy to design and highly flexible, but there is always the temptation to rely upon fairly complex instructions that would not translate to a high performance pipelined implementation[1]. In this chapter we motivate and describe the various design decisions that went into the Monsoon prototype development.

6.1 Multiple Processors

The ETS model (intentionally) takes no position on how to partition a machine into multiple processors. The model is inherently parallel. Any number of tokens can be processed simultaneously as long as the update of a location's presence state and value is performed atomically.

[1] The Manchester dataflow machine [22] is built around microcoded processing elements. This gave substantial freedom for experimenting with the instruction set. We think it unfortunate, however, that the Manchester effort focused on reducing dynamic instruction counts by developing new complex instructions rather than attempting more aggressive compile-time optimizations.

A very simple partitioning approach is taken in Monsoon. As shown in Figure 6.1, a processing element is associated with a region of storage. Given a token $\langle c.s_p, v\rangle$ we know the processing element on which the match will take place by examining the most significant bits of c. That is, the context c can be decomposed into,

$$c = \text{PE:FP}$$

where PE is the **processing element number** and FP is the **frame pointer** to the activation frame with which the token is associated.

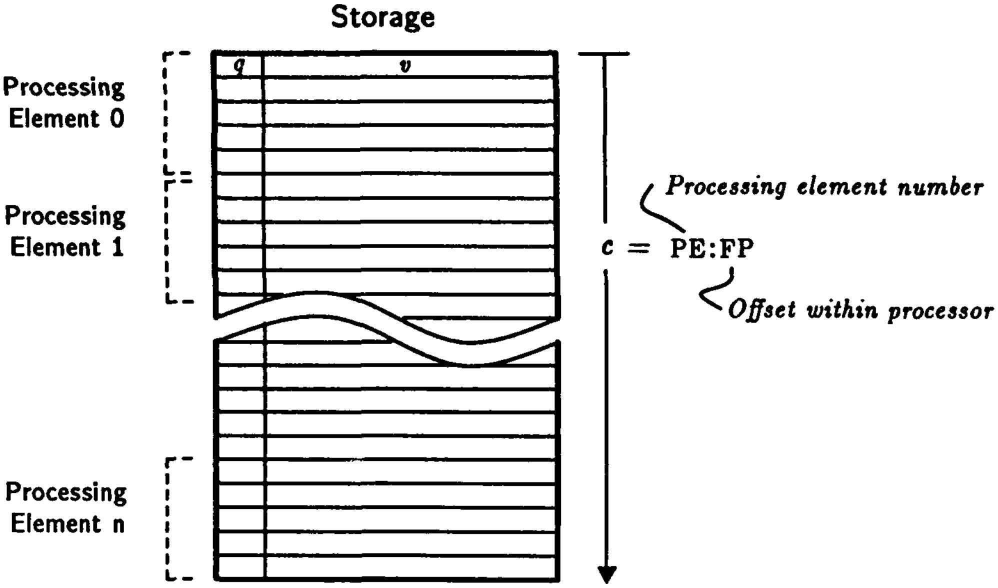


Figure 6.1: Partitioning Storage Across Processing Elements

There is one clear implementation advantage of the approach. If activation frames are prohibited from crossing a processor boundary then token storage can be accessed rapidly and without conflict as private memory within each processor. There is an equally clear disadvantage. Activation frames are assigned to processors at the time `get-context` is processed (*i.e.*, at function application time) and can not easily be relocated dynamically, either to level computation load or to rebalance memory usage among processors.

To achieve dynamic relocation it appears necessary to adopt an address translation mechanism whereby c is dynamically translated into a processing element number and frame pointer (*i.e.*, page translation). Moving an activation frame would involve copying its current contents to another processor and then updating the virtual memory map so that tokens are redirected to the new processor. Activation memory would still be local to a processor.

Even more aggressive would be the divorcing of processors from memory altogether. In this scenario, activation frames would be cached by individual processors. Moving an activation would no longer require the copying of an activation frame, but this scheme introduces the serious challenge of maintaining coherence among the processor caches[2].

However, the need to perform work load leveling is not as clear cut for a machine which supports lots of fine grain tasks as compared to one where tasks have to be large [35]. The code-block invocation provides an expedient boundary for partitioning a computation across a machine. Maa [34] argues that there are enough such invocations in most programs to yield a sufficient number of tasks for distribution among many processors. As long as there is sufficient work to keep processors from idling, there appears to be no reason to average this work across processors. In essence, we are willing to invest some parallelism, by making sure that we give each processor *more than* enough work, in order to avoid workload redistribution. This is similar in spirit to the argument for using parallelism to mask large memory latencies [12].

6.1.1 Sharing Code

In the ETS storage model the statement number s and the context c are both global addresses into shared storage. Even if the address space is divided up across processors so that activation frame references are always local, it is still possible that s would reference an instruction that is not local. It would be relatively straightforward to have instruction caches on each processor, but we have chosen to *require* that instruction references be local. That is,

$$s = \text{PE:IP}$$

where IP is the **instruction pointer** to a location on processing element PE—the *same* processing element implied by c. Thus a complete copy of the code for a code block must exist on every processor on which the code block is executed. We note that the tag fields $c.s$ are replaced by

$$c.s \rightarrow \text{PE:(FP.IP)}$$

where,

[2] This is not as bad as the general cache coherence problem since the runtime system is controlling the migration of activations among processors and can *explicitly* invalidate the participating processors' caches. But there remains the non-trivial problem of maintaining consistency of the translation tables across *all* of the processors.

PE:FP Is the global address of the base of the activation frame in which the match will take place.

PE:IP Is the global address of the destination instruction.

Thus the activation frame slot and destination instruction are both local non-blocking references on processor PE. All we are really doing is trading design simplicity for additional memory, although the encoding does help to reduce the size of the tag from when c and s are independently specifiable.

6.1.2 Interleaving Structures

One disadvantage of the address space partitioning shown in Figure 6.1 concerns the allocation of large data structures. We do not attempt to preserve locality of structure references (*i.e.,* we do not try to keep structures "near" the producers and consumers). Network traffic and processor load can be more evenly distributed by **interleaving** or *spreading out* structures across processors.

When interleaving is performed word-by-word (*i.e.,* element i on processor k, element $i+1$ on processor $k+1$, *etc.*) the memory map is exactly *opposite* of Figure 6.1. Instead, the processing element number should comprise the least significant bits of c,

$$c = \text{FP:PE}$$

We merge these disparate views by conceptually dividing up the address space into regions where *increments* to c advance either within a processor or across processors. We emphasize the incremental nature of our algorithm. Physically c is always of the form $c = \text{PE:FP}$ but *changes* to c (*e.g.,* as performed by **form-address** in Section 4.4) may affect PE more "rapidly" than FP. There are therefore never any aliases for the same location and a bit-for-bit comparison establishes equality of two pointers.

The interleaving pattern can be generalized into **subdomains** [11]. A subdomain is collection of 2^n logical processors starting on a modulo 2^n processor number boundary. If $n = 0$ then increments to c map onto the same processor (*i.e.,* addressing within an activation frame). If $n = 1$ then increments to c alternate between two processors. Figure 6.2 illustrates address interleaving as a function of n for subdomains of size one (a single processor), two and four.

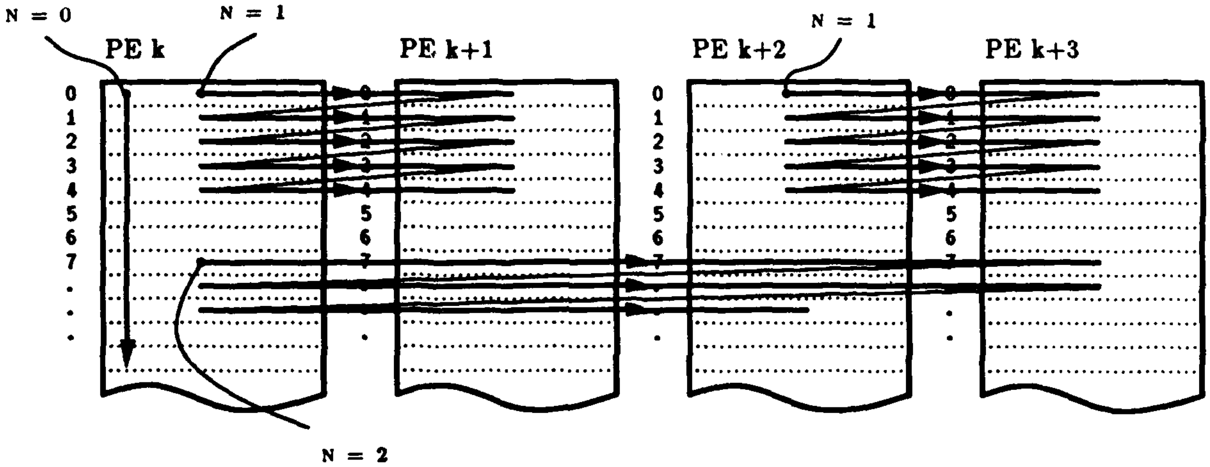


Figure 6.2: Address Interleaving as a Function of n

6.1.3 Two Ways of Referencing Presence Bits

Recall that every storage location has a small number of presence bits (the q in $q.v$). This suggests that presence bits should be physically associated with each memory location. For efficiency, a resource manager might want a very different view. For example, freshly allocated activation frames and I-structures are assumed to have all of their presence bits reset to *empty*. It would be convenient if a resource manager could reset a large number of presence bits with a single instruction.

We have chosen to coalesce presence bits for adjacent locations into words equal to the machine word size. For Monsoon's word size of 64 bits and with two presence bits per location, a word comprises 32 sets of presence bits.

6.2 A LIFO Token Queue

The ETS gives the compiler explicit control over the matching resources required to execute a program. However, the compiler still has little control over the execution order or scheduling of instructions. It is assumed that the tokens generated by an instruction are placed into an automatically managed and fair (*e.g.*, first-in, first-out or FIFO) token queue. There are several reasons why we may want more than FIFO control over the scheduling process:

- **Controlling Parallelism.** A fair token scheduler biases the unfolding of the task tree (see Chapter One) to be breadth-first. This is a fine strategy unless the task tree grows large enough to deplete machine resources, notably memory. A depth-first unfolding of the task tree *tends to* reduce resource requirements[3]. Arguably, this degree of control can (and should) be better exercised by a resource manager and/or a compiler through the introduction of artificial data dependences [18].
- **Improving Locality.** Even with guaranteed-local activation frame and code references, a high performance processor will want to employ both instruction and code caches. The scheduling of tokens can have a major effect on cache locality, where a FIFO token queue is apt to elicit the *worst* behavior. Furthermore, the actual FIFO queue itself cannot be cached.
- **Priority Scheduling.** Priority may want to be given to those tokens which are part of a critical section so as to increase the throughput of (and thus decrease waiting for) resource managers. The need for priority processing could be readily encoded into the destination list of instructions.

Monsoon offers a degree of scheduling control through a priority mechanism and the use of a last-in, first-out (LIFO or stack) token queue rather than a FIFO. The priority of a token is encoded in the destination list of an instruction and specifies either,

1. **Direct Recirculation.** This is the highest priority token; the token queue is bypassed entirely and the token is submitted to the instruction fetch stage on the very next pipeline cycle. The are two obvious restrictions: only one such token is permitted from an instruction and the token has to map onto the same processor (*i.e.*, c does not change). Unless there is an incoming token from the network (see below), one of the result tokens from an instruction will, by default, be directly recirculated. *Uninterruptible* direct recirculation is essential to the correct processing of exceptions, as we shall show in a moment.
2. **A Token Queue Number.** Monsoon has two LIFO token queues that have a strictly ordered priority—tokens are processed from the lower priority queue only when the higher priority queue is empty. Most instructions designate the higher priority queue, but tokens can be "set aside" until the processor is not very busy by inserting them into the lower priority queue.

Another option (not taken in Monsoon) is to encode the priority on the token itself as part of the tag. This would be more dynamic than destination encoding but has

[3] If the programming language has non-strict semantics (like Id) then there may be some mutual intertask dependences that require the whole task tree to exist simultaneously before any tasks can terminate.

the added complexity of rules for propagating priorities. For example, if a low priority matches a high priority token what should be the priority of the result tokens?

There still may be issues of fairness in a multiprogrammed environment, where it is quite possible for one program to completely inhibit processing of another by having "buried" a critical token on the stack of a processor saturated by the first program. Monsoon allows a resource manager to direct the flushing of a queue (*i.e.*, only permit dequeuing, referring new tokens to another stack) and/or to instantaneously reverse the priorities of the two stacks (*i.e.*, causing all low priority tokens to be processed before any high priority ones). If we are mainly relying on the LIFO mechanism to preserve locality, then an occasional disruption may be acceptable.

6.3 Exceptions

For a variety of reasons, an instruction may not be able to complete the specified operation on its input data. **Exceptions** in Monsoon can be generated by the following events:

- **Illegal Presence State Transition.** The matching function can signal an illegal state transition, *e.g.*, detecting an attempted double-write of an I-structure location.
- **Arithmetic Exception.** An arithmetic operation could result in an overflow or underflow exception. We also permit instructions to specify exceptions when a result is negative or zero. In the degenerate case, an arithmetic instruction can *always* trap. This serves two purposes: it permits software generated traps and, by decoding all reserved opcodes into a trap instruction, serves as an illegal opcode trap as well.
- **Type Trap.** All Monsoon values have hardware type-tags, used for example to differentiate integers, floating point numbers and tags. An instruction may specify the range of allowable types on a given input port.

The problem of designing both an efficient and correct exception mechanism is manifold. Most challenging are the abilities to easily reconstruct the state of the computation before the exception occurred and the rapid dispatching to the appropriate exception handler. In an ETS the computation state is completely captured by an activity comprising a tag and the left and right values. On every instruction cycle Monsoon records the operands in an special register set. If an exception occurs then no destination tokens are produced, but an uninterruptible directly recirculating trigger token, whose value is the tag that caused the exception, is produced to activate the exception handler. The exception handler can then consult the saved activity to determine the exceptional event (ALU flags saved in order to accelerate the dispatch) and interpret or restart the offending instruction, if desired.

A special problem occurs in a multithreaded pipelined processor such as Monsoon. At any time there are as many independent activities in the pipeline as there are pipeline stages and *any or all* of these activities may cause an exception. If the pipeline has, say, eight stages then we can think of eight interleaved processes sharing the same processor and label each process with an interleave number (0 to 7). There must be eight copies of saved-activity register, indexed by interleave number, and a separate exception handler context for each interleave (possibly all sharing the same code, but a different activation frame).

When an exception handler is entered, the associated interleave's activity register is no longer updated (doing so would destroy the saved state) and an exception occurring during this time would result in a fatal **machine check,** an unrecoverable condition. The exception handler must continue to occupy its interleave in the pipeline so it must specify uninterruptible direct recirculation for all instructions. All but the most trivial handlers will want to acquire a new context and copy the exceptional activity register into it as soon as possible, so as to permit exceptions within the exception handler.

6.4 The Network

A well-engineered network that is matched to the demands of the processors is essential to the success of any general purpose multiprocessor. While it is true that dataflow machines are resilient to network latency, *given sufficient parallelism,* there is no working around insufficient bandwidth[4]. Dataflow machines (with suitable compiling restrictions) can also process tokens in an arbitrary order and would seem to be immune to reordering of

[4] To paraphrase Glen Culler, *"There is no substitute for real bandwidth."* Let the reader not be confused with Seymour Cray's immortal wisdom, *"There is no substitute for real memory."* Let us add ours: *"Being able to tolerate latency is no excuse for introducing it."*

messages by the network. We show an insidious write-acknowledgment problem, however, that will make resource management much more difficult without network support.

6.4.1 Bandwidth Requirements

Because we do not rely on locality of structure references network bandwidth requirements will be high but easy to predict. In fact, the whole purpose of structure interleaving is to distribute network traffic so that the probability of a given structure reference being satisfied locally is low, approximately $1/p$ where p is the number of processing elements in the machine. A conservative assumption is that *every* fetch and store generates network traffic.

The percentages of fetches and stores in scientific programs written in Id and compiled for the TTDA are similar to the dynamic mixes of fetches and stores in conventional machines, about 30–40% [8]. Thus, on the average, about every other instruction will generate a token for the network (including tokens for function arguments). If a pipeline has a sustained instruction rate of k instructions per second then the network must have a *sustained bandwidth of $k/2$ tokens* per second per network port. These are tokens *sent* by processors, and for every token sent there must be one received. The total number of tokens entering and leaving the processor will therefore average k tokens per second.

In Monsoon, tokens are 144 bits, or about 200 bits when including network overhead (routing header, cyclical redundancy checks, *etc.*). A pipeline that sustains ten million instructions per second will place a *sustained* demand on a network port of,

$$200\frac{k}{2} = 1 \; gigabit/second$$

For comparison, a single processor CRAY X-MP has a main memory bandwidth of about 15 gigabits/second.

6.4.2 The Write-Acknowledgment Problem

In the ideal TTDA model, the only instruction ordering constraints are based upon the availability of data, and a (terminating) program will provably yield the same result independent of the order in which tokens are processed. As soon as resource management enters the picture, unexpected problems can arise. Recall from Chapter Two that TTDA code blocks are compiled to produce a signal token upon termination—meaning that all zero-output instructions in the code block have fired. So zero-output instructions such as `i-store` are designed to emit signals *when the instruction fires*, which are then coalesced by a signal tree whose output becomes the termination signal from the code block.

The *firing* of an `i-store` only means that a store request has been sent to the location *not* that the location has *received* the new value. Under normal circumstances this does not cause a problem because reads are synchronized on a location-by-location basis. However, suppose the compiler deduces that the lifetime of a structure is tied to the lifetime of a code block activation (an active research goal). It could then use the termination signal to explicitly deallocate the structure, by sending a deallocate request to a resource manager which subsequently reinitializes the I-structure. Here is the problem: there is race between the deallocation request and any writes to locations in the structure which are never read.

As shown in Figure 6.3, even FIFO ordering of network messages does not solve the problem. The write requests could go from processor A to B, incurring a large delay, while the termination signal is rapidly processed by C which in turn sends a deallocate request to B before the write arrives. The solution to this problem in Monsoon is to inhibit the generation of a signal from an `i-store` until the message has been *received* by the destination processor. This is accomplished by a special **circuit switched** message transmission mode in which the network maintains a back channel for acknowledgment of message receipt. We do not block the pipeline when an `i-store` is processed (a sure formula for deadlock) but set the signal aside and reintroduce it when the acknowledgment is finally received.

This solution also requires incoming network messages to be processed in strict FIFO order, as *receiving* a store request still does not mean the store has taken place. By processing network messages in a FIFO order, however, we can be assured that the deallocate request can not get in front of the write. Because the token queues on Monsoon are not FIFO, network messages are inserted directly into the pipeline. To avoid deadlock, an incoming network token must have the highest priority (except for uninterruptible directly recirculating tokens).

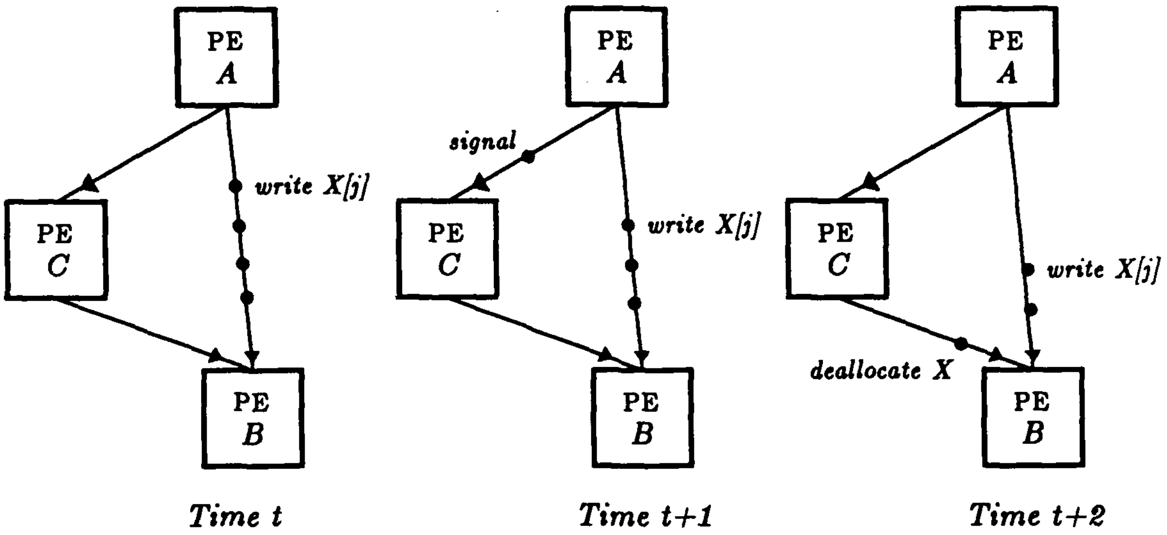


Figure 6.3: A Race Between an `i-store` and a Deallocate

6.5 Hardware Support for Infinite Processor Simulation

Simulating the execution of a dataflow graph on an infinite processor machine allows one to characterize the parallelism inherent in a program [13] [46]. Rather than simulating against time, the simulation is performed on a **computation step:** all tokens generated during step j are processed before proceeding to step $j + 1$. The total number of steps is the **critical path** of the program: *no* machine would be able to compute the same graph in fewer steps.

We will use the two token stacks to perform an ideal time step simulation with great efficiency. As shown in Figure 6.4 at computation timestep j all of the tokens are in stack 0. The processor, at full speed, dequeues tokens from stack 0, processes them, and enqueues the new tokens on stack 1. The statistics counters are read and then the roles of stack 0 and stack 1 are reversed. The process continues until there are no tokens remaining.

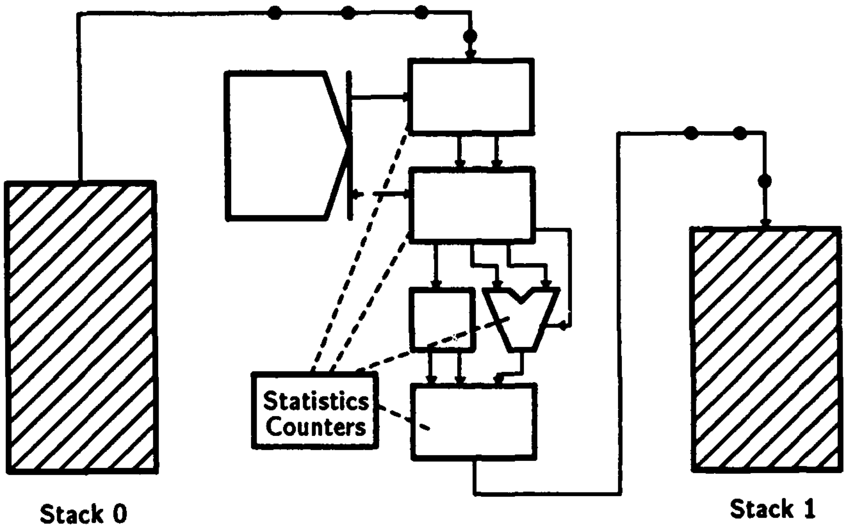


Figure 6.4: Performing An Ideal Computation Timestep Using Two Stacks

6.6 Machine Data Types

Monsoon has a fixed word size of 72 bits—64 bits of data and 8 bits of type-tag. In this section we show how the various machine data types are represented on Monsoon. We have chosen a 64-bit data values for several reasons, including the desire to make Monsoon a "serious" machine for scientific applications that require full precision floating point arithmetic. Another important motivation for 64-bit words is to provide an adequate address space. Remember that ETS requires a fixed word size large enough to store a tag, and that a tag encodes two addresses (the frame pointer and the instruction pointer). The address space is therefore about *half* of the word size, in this case 32 bits. A word size of less than 64 bits would result in an unacceptably small address space.

6.6.1 Tokens

A token is a 144 bit quantity comprising a tag-part and a value-part. The tag-part and value-part are each 72 bits—8 bits of TYPE and 64 bits of VALUE, as follows:

Token			
Tag-part		Value-part	
TYPE 8	TAG 64	TYPE 8	VALUE 64

A TAG is a kind of VALUE and the tag-part must have a value of type TAG. A VALUE can be one of four machine data types:

Machine Data Types		
Type	Representation	Example Operations
TAG	Tag	Increment, Field Set and Extract
FLT	IEEE Double Precision	Add, Sub, Mult, Compare, Convert
INT	64 Bit Integer	Add, Sub, Mult, Shift
BITS	64 Bit Unsigned Integer	16 ALUs, Bit Rotate

There is no hardware-defined meaning of the TYPE fields. The hardware does not check for the ᵔepresentation validity for any operations[5]. For example, this allows instructions to manipulate a tag using boolean operations. The interpretation of type-tags is completely programmable through type checking and propagation hardware to be described in a moment.

Tags

A tag has the following format:

TAG				
PORT	MAP	IP	PE	FP
1	7	24	10	22

where,

PORT Indicates the l ($= 0$) or r ($= 1$) port of the instruction specified by PE:IP.

MAP Alias and interleave control. Increments to FP affect PE as specified by MAP.

IP Instruction pointer. The absolute address of an instruction on processor number PE.

PE Processing element number. For machines with less than 1024 physical processors, the LSBs of PE can be concatenated with the MSBs of FP, extending the physical address space of each PE

[5] Floating-point operations will produce exception codes for invalid operand formats, however.

FP Frame pointer. The absolute address of a 72 bit location on processor number PE. PE:FP describes a global address, so a machine is limited to a maximum of 4000 megawords of physical memory.

Tags are also used as pointers (*e.g.*, for referencing I-structures) but **incrementing** a tag (adding an offset to the pointer) can *not* be performed using an integer add, as is common on many von Neumann machines. The map field controls *updates* to the PE number when adding an offset to the FP field. The map implements TTDA subdomains. The lower n bits of PE define the relative processor number within a subdomain whose base processor PE number is obtained by setting these n bits to zero. Interleaving of FP is performed on single word boundaries. The internal structure of the map field is as follows:

MAP	
HASH	N
2	5

where,

HASH Selects the interleave strategy: FP or aliased FP

N Logarithm (base 2) of the number of processors in the subdomain.

HASH is encoded as follows:

HASH Field Encoding		
HASH	Strategy	Use
00	FP	Word interleaved data structures
01	aliased FP	Constant data structures
1x		Reserved

Remember that the map affects PE only when *incrementing (adding an offset to)* a pointer. An example of interleaving FP was shown in Figure 6.2. If HASH = aliased FP then the N least significant PE bits are ignored—any processor in the subdomain can service the token. This allows constant data structures to be replicated in order to even out network and processor loading. Tag equality can still be performed using a boolean compare as long as tags that specify aliasing have the least significant N bits of PE set to zero.

Floating Point Numbers

Floating point numbers conform to IEEE Standard 754 for double precision. Extended precision floating point numbers are not directly supported by the hardware.

Full (Double) Precision Floating Point Number		
SIGN	EXPONENT	MANTISSA
1	11	52

Integers

Integers are 64 bits, unsigned or signed 2's complement notation.

Bits

The BITS datatype is implemented with 64-bit unsigned integers. The complete set of bitwise "ALUs" (all of the sixteen functions on two bits) are provided as well as bit rotates, reversals and shifts. The machine recognizes a FALSE value as a word of all zeros and a TRUE as anything else. If the hardware generates a boolean value (say, from a compare operation) then TRUE is the word of all ones.

6.6.2 Instructions

An instruction is 32 bits wide. The local memory is 72 bits wide, large enough for two instructions. An instruction can be manipulated as a field of bits using boolean operations. By convention, we assume that the presence bits on all locations that contain instructions are set to *present*, so that literal constants may be stored as part of the program text.

Because we have packed two instructions into a word, the least significant bit of IP is not part of the address to local memory but is a selector for the upper or lower halfword[6]. The instruction encoding is simple, consisting of just four fields:

[6] If the LSB of IP = 0 then the instruction is obtained from the least significant halfword of location IP SHR 1.

Instruction			
OPCODE	r	PORT	s
10	10	1	11

where,

OPCODE The instruction opcode. OPCODE is decoded by table lookup into an effective address control (E), a token matching rule (W), an ALU operation (A) and a token forming rule (T).

r An unsigned offset used to compute the effective address of the operand slot.

PORT Defines the l (= 0) or r (= 1) port for one of the destination tags.

s A 2's complement offset to IP for one of the destination tags.

Note that a Monsoon instruction specifies only *single* destination (PORT and S) whereas the ETS calls for two destinations. As an encoding technique, one of the destinations is always IP + 1 with PORT = l.

6.7 Processing Pipeline

The Monsoon pipeline processes tokens at a rate of one per pipeline cycle[7]. As shown in Figure 6.5, an incoming token either is recirculated from the pipeline output, is popped from a token stack, or comes directly from the network. A decoded instruction tracks an incoming token through each stage, performing local operations, ultimately producing a new token(s) that either is directly recirculated, pushed onto a token stack, or sent over the network.

A token's trip through the pipe can involve at most one operand from the local memory. Typically the operand is a slot in an activation frame. Each word in local memory is 72 bits wide, large enough to hold a tag, a value, or two instructions. In addition, two presence bits are associated with each local memory word.

[7] We distinguish a pipeline cycle from a clock cycle. Sometimes it will be necessary, for instance during an exchange, to clock a pipeline stage twice.

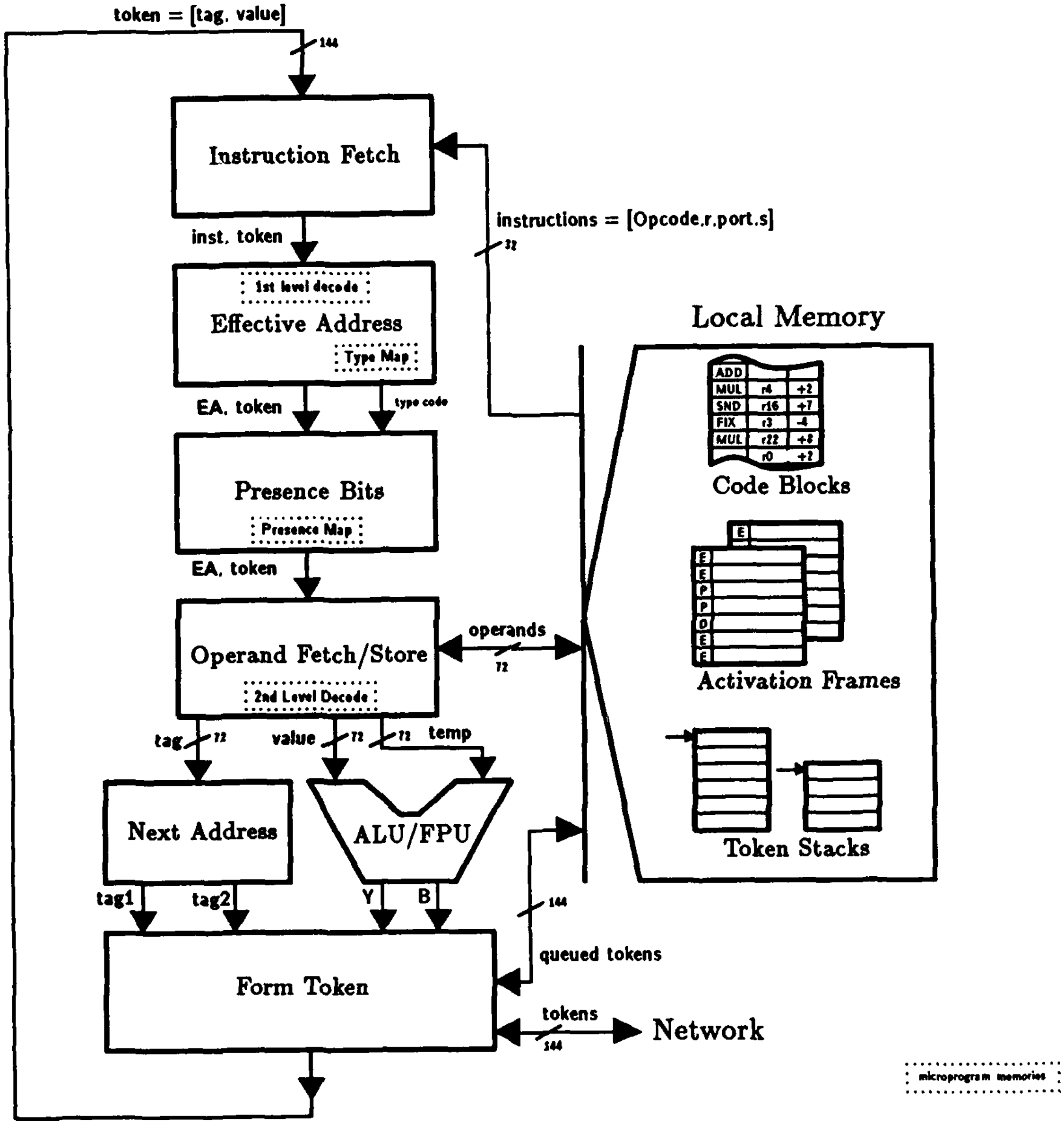


Figure 6.5: Monsoon Processing Pipeline Overview

The Monsoon processor pipeline consists of six pipeline stages; instruction fetch, effective address calculation, presence bits, operand fetch/store, Next Address and FPU/ALU, and form token. A fetched instruction is subsequently decoded into "horizontal" control points for each stage. This decoding is table-driven, the opcode is used as an address into decoding memories. The basic functionality of the stages is as follows:

1. **Instruction Fetch.** The IP field of the incoming token is treated as an absolute address into local memory. A single 32 bit instruction is fetched.

2. **Effective Address.** An effective operand address is computed by one of several modes: EA =FP+r, IP+r, or simply r. Simultaneous type analysis is performed on the VALUE TYPE field. This is accomplished by a **type map**, a table lookup on the TYPE field (eight bits) concatenated with the incoming PORT bit, resulting in a two bit type dispatch code, TC.

3. **Presence Bits.** The two presence bits for location EA are operated upon. This is accomplished by a table lookup on the concatenation of the current state (two bits), the incoming PORT bit, and the type dispatch code TC (two bits). A finite state machine issues a new state for location EA, an opcode for the operand fetch/store stage, a four-way decode branch control, and a force-opcode-to-zero override.

4. **Operand Fetch/Store.** As directed by the presence bits state table output, the memory location at EA is either read, written with the VALUE part of the token, or atomically exchanged with the VALUE part. During either a read or an exchange, the contents at address EA are passed to the ALU/FPU as *temp*.

5. **ALU/FPU.** The ALU/FPU operates upon the VALUE part of the token and *temp*, if applicable. In the case of a dyadic operation the hardware sorts out the left (L) and the right (R) operand from the incoming token and *temp* and presents them to the ALU/FPU as $l \rightarrow A$ and $r \rightarrow B$. Additionally, the instruction decode may call for a "cross-over", $l \rightarrow B$ and $r \rightarrow A$. Two results are produced: $Y = A$ OP B and B. In those cases where *temp* is not generated the hardware forces $A = B$. The ALU/FPU also performs type propagation and exception and condition code masking and generation.

6. **Next Address.** The next address unit operates in parallel with the ALU/FPU. It operates on the TAG part of the token whereas the ALU/FPU operates on the VALUE part. Specifically, the next address unit operates on the PORT and IP field (and implicitly the PE field depending upon MAP) to generate two new tags. The first tag is obtained by adding the s part of the instruction to the current IP and substituting the instruction-specified PORT. The second tag is generated by incrementing the current IP by either $0, +1, +2$ or $+3$ and by forcing the output port $= l$.

7. **Form Token.** The two values produced by the ALU/FPU and the two tags produced by the next address unit are conditionally assembled into zero, one or two tokens. The form token stage also manages ιhe token stacks and arbitrates the network connection.

Instruction decoding is controlled by the contents of four tables (refer to Figure 6.6). The detailed field encoding assignments are documented in Appendix A.

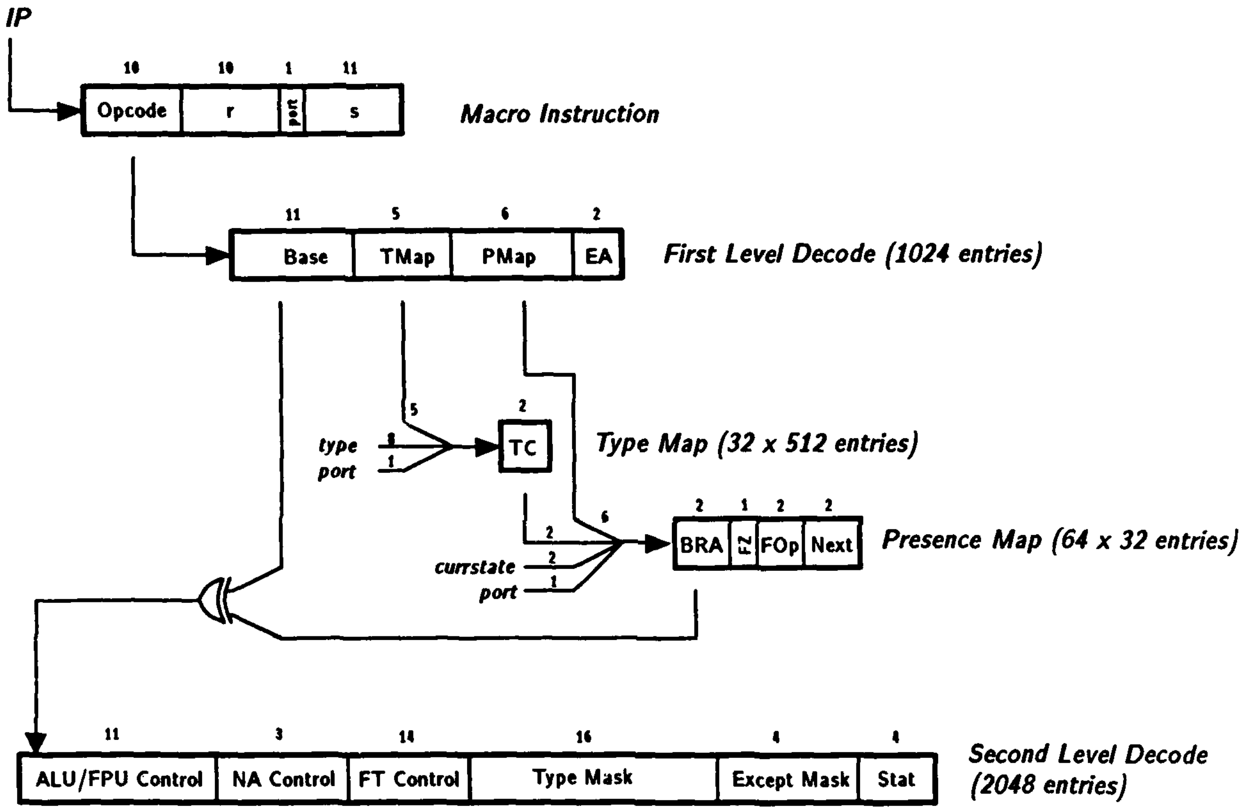


Figure 6.6: Instruction Decoding Tables and Maps

1. **First Level Decode.** Specifies the effective address generation mode, the type and presence maps, and the base address for the second level decode. The macro instruction opcode is the address for the first level decode table, so there are $2^{10} = 1024$ entries of 24 bits each.

2. **Type Map.** A lookup on the TYPE and PORT of the VALUE part of the incoming token, yielding a two bit type dispatch code, TC. There are 32 type maps each with 512 two bit entries.

3. **Presence Map.** Maps the current presence state of address EA into a new state. The incoming PORT and two-bit type code TC are additional inputs. The presence

map also specifies the frame store operation (read, write, or exchange) and one of eight branch points into the second level decode. There are 64 maps of 32 seven-bit entries.

4. **Second Level Decode.** Specifies the control for the next address, FPU/ALU, and form token stages. The second level decode address is generated from the logical OR of the base address from the first level decode with the two branch bits from the presence map. If the presence map asserts force-zero, then the base is set to zero and the two branch bits specify the absolute second level decode entries 0, 1, 2 or 3. There are 2048 entries of 56 bits each.

6.8 Status

A single processor version of the Monsoon pipeline has been constructed and is currently being tested for operation at eight million dataflow instructions per second. The design has been implemented with off-the-shelf components (TTL and programmable logic devices) and comprises approximately 500 integrated circuits and consumes under 150 watts. The prototype processor is hosted by an Texas Instruments Explorer Lisp Machine. The purpose of the prototype is a proof-of-concept and an early software development vehicle.

A second version now being designed will employ CMOS gate arrays to reduce the size and power and to improve the cycle time. We expect this version to act as a building block for a machine with 100 to 1000 processing elements. A 256 processor machine would yield a sustained performance of approximately one billion dataflow instructions per second (assumes 50% processor utilization, 60% dyadic instructions), any mix of which may be floating point operations.

We have found the design to be large in terms of gate count (due to the wide data paths) but of manageable complexity. The pipeline control is surprisingly simple, as all pipeline stages are independent and there are no potential hazards or interlocks. We believe that the design will readily scale to much faster implementation technologies.

Chapter 7

Conclusion

The most persuasive arguments for multiprocessors are economic, not academic. It is dangerous to justify multiprocessors as the "only way" to make the highest performance machines. Indeed, todays title for the fastest supercomputer [33] goes to a uniprocessor: the NEC SX-2. There is little doubt that basic circuit technologies, in contrast to architectural innovation, will continue to be the dominant driving force behind ever-faster and cheaper processing elements. History has uniformly disproved those who predict fundamental limits on uniprocessor performance.

At any time, there is a wide spectrum of implementation technology choices, ranging from commodity ("jellybean") to exotic, and the first-order difference in cost/performance among contemporary uniprocessors is a function of the underlying circuit technologies. However, there is by no means a constant or even linear relationship between cost and performance—doubling performance may more than double the cost: the law of diminishing returns. Figure 7.1 illustrates the typical cost–performance tradeoff as a plot of cost per operation per second versus total operations per second. The performance is measured by the LINPACK benchmark [29] and is given in megaflops, or millions of floating point operations per second. We do not mean to imply that LINPACK is a good predictor of performance across a broad range of applications; we use it only to illustrate the typical marginal cost of increased performance.

Supercomputers, by definition the fastest machines that can built at any time, offer the highest performance but at the highest cost per unit performance. At the other extreme, the low-end personal computer system is dominated by a the cost of the keyboard, power supply and storage devices and offers poor cost/performance.

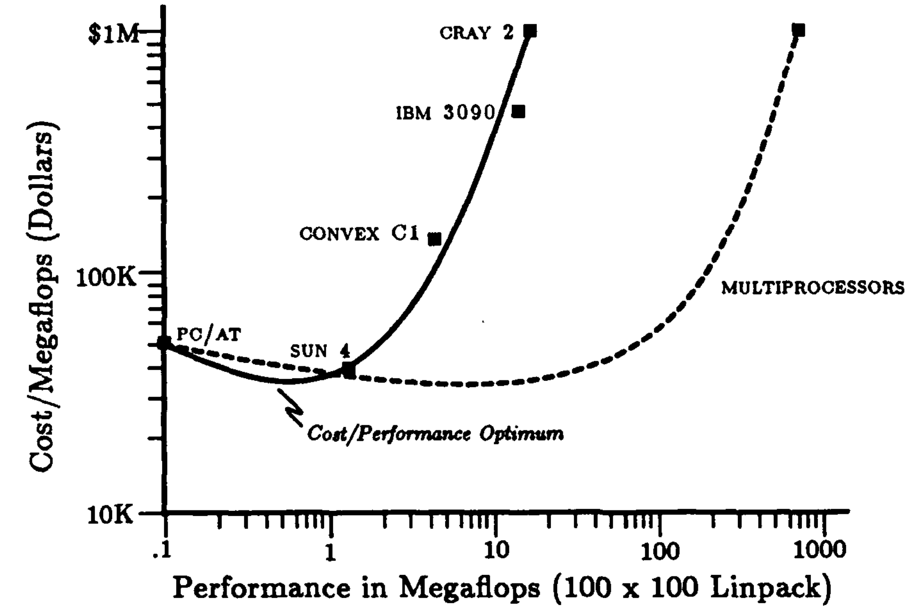


Figure 7.1: The Cost–Performance of Uniprocessors and the Multiprocessor Promise

The real economic opportunity for multiprocessors is represented by machines at cost/performance optimum. As suggested by Figure 7.1 a multiprocessor constructed from these machines could be over an order of magnitude less expensive than an equivalent-performance uniprocessor supercomputer. At any time, the effect of successful multiprocessing is the flattening out and movement to the right of the cost/performance curve. That is, most every machine will be built from the most cost-effective technology. And the "performance at any cost" machines will be (very expensive) multiprocessors built from exotic technology.

Today, there is no shortage of multiprocessors that cost one-tenth of a Cray nor a shortage of salespeople who will demonstrate an application on the machine that meets or exceeds the performance of a Cray. But there *does* seem to be a shortage of people wanting to buy these multiprocessors. The problem, of course, is that the cost/performance of a machine on a benchmark does not reflect the cost of getting the program to run in the first place. The *programming cost* for the *effective utilization* of a multiprocessor can be enormous [44]. It may often be the case that the program cannot be made to run faster no matter how much programming effort is invested. Then the multiprocessor will apt offer *far worse* cost/performance compared with the Cray.

The promise of dataflow machines is that they will be far easier to use. The hardware-enforced fine grain synchronization simultaneously makes compiling easier while exposing the maximum amount of parallelism. Dataflow hardware *is* (presently) more costly than a von Neumann processor of equivalent peak performance. The goal of thesis was to try to reduce the complexity, and thus the cost, of tagged token dataflow machines. We believe that we have largely succeeded. The explicit token store addresses several fundamental "implementability" concerns of tagged token dataflow machines: the associative matching store, the large set of context registers, the need for separate resource manager processors, and some fairly complex instructions and specialized machine data types (*e.g.,* I-structure descriptors). At the same time, we have shown that the complexity can be reduced without affecting performance or expressiveness.

In spite of all this, we believe that tagged token dataflow machines are a ways from commercial viability as general purpose multiprocessors. We close with a charge to those wishing to bring dataflow architectures to the mainstream of computing:

1. Compile conventional languages like COBOL and FORTRAN for a dataflow machine such that a maximum amount of parallelism is exposed while preserving deterministic execution.
2. Discover resource management policies that can govern the tremendous amount of parallelism uncovered by the hardware. Develop highly efficient storage allocation and reclamation, perhaps automatic garbage collection.
3. Improve the basic cost/performance of the processing element. Understand the behavior of caches for local and non-local references and the benefits of "three-address" and multiple operations per instruction. Achieve dynamic instruction efficiencies that meet or exceed von Neumann processors.
4. Develop scalable input/output systems of commensurate performance.

The ultimate goal seems clear: even a *single processor* dataflow machine should outperform an equivalent-cost conventional uniprocessor.

Appendix A

Monsoon Instruction Decoding

Monsoon is a fully pipelined processor, but for experimental flexibility Monsoon makes extensive use of table-driven decoding of instructions for all pipeline stages. As such, the processor can be "customized" to a particular instruction set. For example, the Monsoon processor supports hardware type-tags. The decoding tables allow an instruction set to be generated which at one extreme supports "generic" or "auto-coercive" operators, while at the other extreme completely ignores hardware type-tags. The objective of this appendix is to describe the **microarchitecture**, the encoding of the various control fields, of Monsoon in detail. Please refer to Chapter Six, Figure 6.6 for a relationship among the various decoding fields.

A.1 First Level Decode

The instruction OPCODE is used as the address for the first level decode. The first level decode specifies the effective address generation mode, the type and presence maps, and the base address for the second level decode.

First Level Decode Entry			
BASE	TMAP	PMAP	EA
11	5	6	2

where,

BASE Specifies the base address for the second level decode.

TMAP Specifies one of 32 type maps.

PMAP Specifies one of 64 presence maps.

EA Specifies the effective address generation mode.

A.1.1 Effective Address Generation

The operand address is calculated by the effective address unit. The effective address generation always involves adding r to a masked version of FP or IP. Refer to figure A.1

The are three encodings of the two EA bits as follows:

EA Field Encoding		
EA	Expression	Use
00	FP $+r$	Activation frame relative (operands)
10	IP $+r$	Instruction relative (literals)
11	r	Absolute (system)

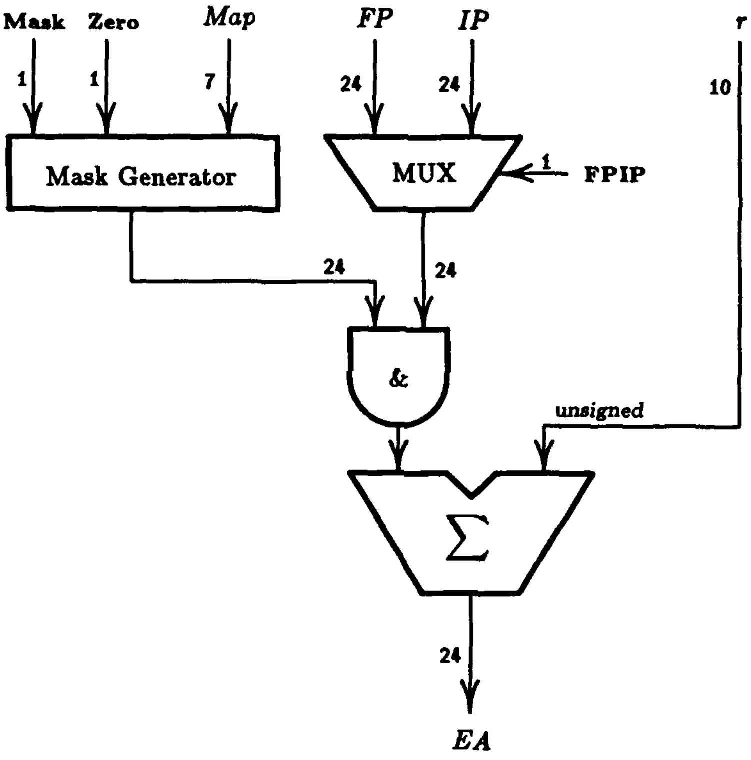


Figure A.1: Effective Address Generator

These correspond to the effective address mode specified by the ETS *except* that the absolute mode does not refer to a global storage location but is an absolute address in the processor's local memory.

A.1.2 Type Map

The type map takes the VALUE TYPE and PORT and maps them into a two bit type code. The type code is used as input to the presence map.

The TMAP field selects one of 32 type maps. A type map is conceptually a two dimensional array of 256 types × 2 ports. This means that each map has $2^9 = 512$ entries of two bits each. There are no preassigned meanings to the bits, they are purely programmer convention.

Type Map		
TYPE	PORT = l	PORT = r
00000000	$TC_{0,0}$	$TC_{0,1}$
00000001	$TC_{1,0}$	$TC_{1,1}$
00000010	$TC_{2,0}$	$TC_{2,1}$
...	...	...
11111110	$TC_{254,0}$	$TC_{254,1}$
11111111	$TC_{255,0}$	$TC_{255,1}$

The power behind the type map is that it can affect the presence state transition. That is, presence state machine has TC as inputs in addition to the current state and incoming port. The presence state machine not only updates the current state but can select from up to *eight* possible decodes for the instruction in the remaining pipeline stages. Thus, the type of the value-part of an incoming token can directly affect the operation performed.

A.1.3 Presence Map

The presence map takes the two presence bits at location EA, the incoming PORT, and the two type code bits TC and maps them into two new presence bits, a frame store operation, and a second level decode branch point. The PMAP field selects one of 64 presence maps. A presence map is a state transition table from the current presence to the new presence state. PORT and type code act as inputs. The frame store operation and branch code are outputs. Each presence map entry, PENT has the following fields:

PENT			
BRA	FZ	FOP	NEXT
2	1	2	2

where,

BRA Four-way branch control for the second level decode. The two BRA bits are ORed into the two least significant BASE bits (BASE is a first level decode field).

FZ Force-to-zero override. When asserted (1) the BASE field is forced to zero *before* the ORing of BRA. This yields four absolute second level decode entries in addition to the four base relative entries, or a possible eight-way branch.

FOP Specifies the operand fetch/store operation, defined below.

NEXT. The new value of the presence bits associated with location EA.

A presence map consists of $2^{1+2+2} = 32$ entries as follows:

Presence Map

Inputs		Current State			
PORT	TC	00	01	10	11
l	00	$\text{PENT}_{0,0}$	$\text{PENT}_{0,1}$	$\text{PENT}_{0,2}$	$\text{PENT}_{0,3}$
l	01	$\text{PENT}_{1,0}$	$\text{PENT}_{1,1}$	$\text{PENT}_{1,2}$	$\text{PENT}_{1,3}$
l	10	$\text{PENT}_{2,0}$	$\text{PENT}_{2,1}$	$\text{PENT}_{2,2}$	$\text{PENT}_{2,3}$
l	11	$\text{PENT}_{3,0}$	$\text{PENT}_{3,1}$	$\text{PENT}_{3,2}$	$\text{PENT}_{3,3}$
r	00	$\text{PENT}_{4,0}$	$\text{PENT}_{4,1}$	$\text{PENT}_{4,2}$	$\text{PENT}_{4,3}$
r	01	$\text{PENT}_{5,0}$	$\text{PENT}_{5,1}$	$\text{PENT}_{5,2}$	$\text{PENT}_{5,3}$
r	10	$\text{PENT}_{6,0}$	$\text{PENT}_{6,1}$	$\text{PENT}_{6,2}$	$\text{PENT}_{6,3}$
r	11	$\text{PENT}_{7,0}$	$\text{PENT}_{7,1}$	$\text{PENT}_{7,2}$	$\text{PENT}_{7,3}$

Out of the 64 possible maps defined by PMAP, the first sixteen have reserved meanings as indicated by the following table.

Special Presence Maps

PMAP	Additional Functionality	
0–3	*NOP*	No frame store operation (FOP ignored)
4–7	*Bulk*	Presence bits of adjacent (modulo 32) words all updated
8–11	*Write-Through*	All operand writes update main memory
12–15	*Non-Cacheable*	Reads and writes avoid operand cache

The actual value of the presence bits have no hardware significance *except* 00. This state is a distinguished **empty state,** the associated word has an arbitrary (and irrelevant) value. This is used in the cache roll-out algorithm—values of words in the empty state are never written back to memory *even if they are dirty* (although the presence bits must be written back).

A.1.4 Operand Fetch/Store

The operand fetch/store stage performs a read, write, or exchange on local memory location EA. In the case of a write or exchange the quantity written is simply the 72 bit VALUE field of the incoming token. In the case of a read or exchange the stage produces an additional 72 bit result called *temp*, which is the contents of $[EA]$. Otherwise $temp =$ VALUE. The operation performed is determined by FOP as follows:

FOP Field Encoding

FOP	operation	
00	*Read*	$temp \leftarrow [EA]$
01	*Write*	$[EA] \leftarrow$ VALUE
10	*Exchange*	$temp \leftarrow [EA]$; $[EA] \leftarrow$ VALUE
11	*Decrement-Exchange*	$temp \leftarrow [EA]$; $[EA] \leftarrow$ (VALUE.IP - 1)

Note that if PMAP $= 0, 1, 2, 3$ then FOP is preempted, *no* memory operation takes place and *temp* is set to the incoming token VALUE field.

A.2 Second Level Decode

The second level decode address is computed as (BASE OR BRA) where BASE is the base entry point from the first level decode and BRA is the two bit branch code from the presence map. If the presence map asserts FZ (1) then BASE is forced to zero and the second level decode address is simply BRA, absolute locations 0,1,2 or 3.

The second level decode specifies the ALU/FPU opcode, controls next address generation, and specifies the form token mode. In addition, the second level decode controls type propagation, condition and exception masking, and statistics gathering.

Second Level Decode Entry

FUCTL	NACTL	FTCTL	TMASK	EMASK	STATS
11	3	12	16	10	4

where,

FUCTL Selects a function unit and specifies its control.

NACTL Controls the next address generation.

FTCTL Specifies the form token mode.

TMASK Specifies operand type checking and propagation.

EMASK Specifies the exception mask.

STATS Specifies an increment for one of 16 instruction mix counters.

A.2.1 Function Unit Control

The function units operate on the *temp* and VALUE fields produced by the operand fetch/store stage. First, *temp* and VALUE are resolved into a pair of input operands A and B in relation to the incoming PORT. The A and B operands are fed to four function units, only *one* of which is selected by the current microinstruction. The active function unit produces a result, Y, a delayed version of B and condition and exception codes. The Y and B outputs are sent to the form token unit. All of this is under control of FUCTL, which has the following structure:

FUCTL		
FLIP	UNIT	OP
1	2	8

where,

FLIP Specifies the *l*,*r* → A, B mapping.

UNIT Selects one of the four function units: FALU, PIU, MCU or TPU.

OP Function unit opcode. Interpreted by each function unit.

A schematic of the function unit interconnection is given in Figure A.2. The FLIP bit determines how *temp* and VALUE are to mapped to A and B. This is a function of the incoming PORT bit as follows:

Determination of A and B

	PORT	
FLIP	*l*	*r*
0	$A \leftarrow$ VALUE $B \leftarrow$ *temp*	$A \leftarrow$ *temp* $B \leftarrow$ VALUE
1	$A \leftarrow$ *temp* $B \leftarrow$ VALUE	$A \leftarrow$ VALUE $B \leftarrow$ *temp*

So, regardless of how it happens, FLIP specifies the following relationship between *l*,*r* and A, B.

Encoding of FLIP

FLIP	Mapping
0	$A \leftarrow l$ $B \leftarrow r$
1	$A \leftarrow r$ $B \leftarrow l$

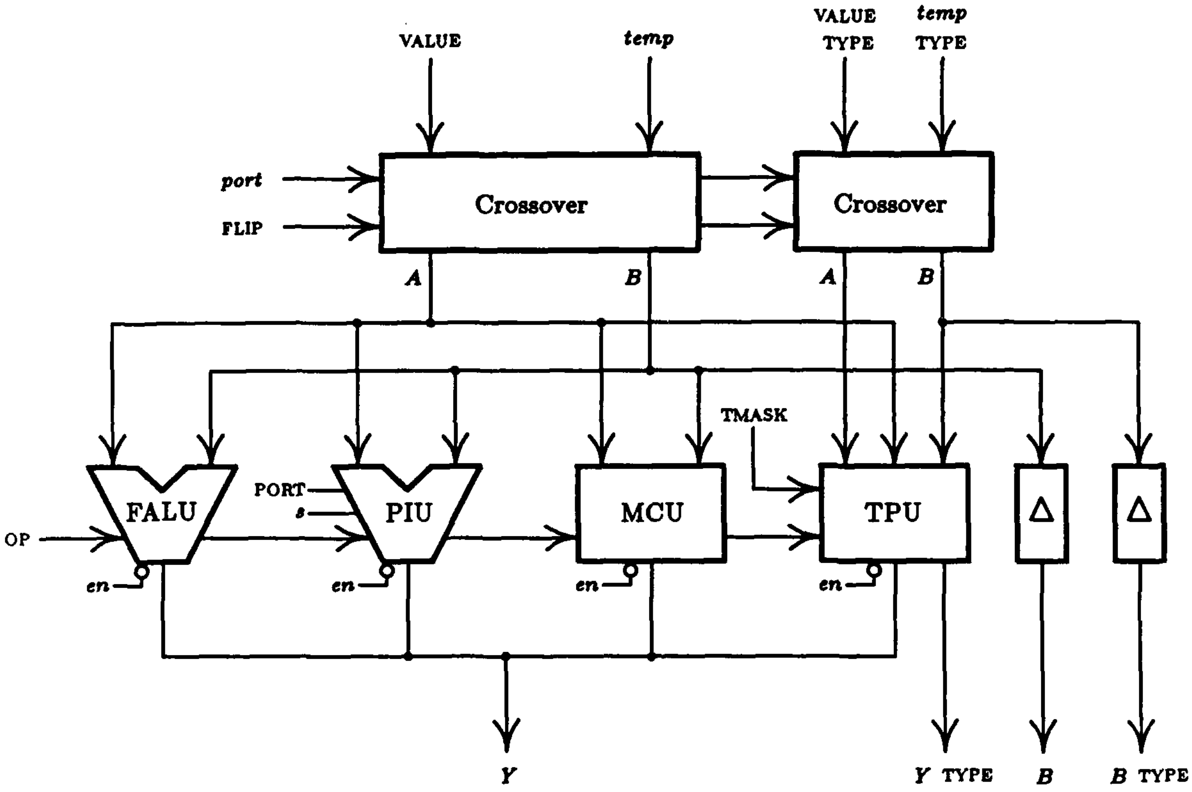


Figure A.2: Function Unit Interconnections

The UNIT field selects one of the four parallel function units:

Encoding of UNIT		
UNIT	Function Unit	
00	*FALU*	Floating point, arithmetic and logic unit
01	*PIU*	Pointer increment unit
10	*TPU*	Type propagation unit
11	*MCU*	Machine control unit

A.2.2 Floating point, arithmetic and logic unit

The floating point, arithmetic and logic unit provides hardware support for floating point numbers, integers, and booleans. For brevity, the OP field encoding is omitted. The kinds of operations are similar to those found on any machine which supports floating point and integer arithmetic.

A.2.3 Pointer increment unit

The pointer increment unit (PIU) computes updates to a pointer's IP, FP and PORT fields, and as a side-effect PE as directed by MAP. The PIU assumes that the *A* operand is of type TAG. The *B* operand is assumed to be a signed integer when it is involved in the update. The updates to the IP, FP and PORT fields are independently controlled by OP.

OP when UNIT = 01			
UNDEFINED	PORTOP	IPOP	FPOP
2	2	2	2

where,

PORTOP Controls the generation of the PORT field.

IPOP Controls the generation of the IP field.

FPOP Controls the generation of the FP field.

Note that the resulting MAP field is identical to the *A* MAP field, and that PE may change as determined by MAP. PORTOP, IPOP and FPOP are encoded as follows:

PORTOP Field Encoding

PORTOP	Resulting PORT
00	A PORT
01	instruction PORT
10	l
11	r

IPOP Field Encoding

IPOP	Resulting IP
00	A IP + B
01	A IP + s
10	A IP
11	B

FPOP Field Encoding

FPOP	Resulting FP
00	A FP + B
01	A FP + s
10	A FP
11	B

A.2.4 Type Propagation Unit

The type propagation unit controls the generation of the Y TYPE field. This involves two second level decode fields. TMASK is always active and normally used to control the generation of the Y TYPE. Alternatively, when UNIT = 10 the OP field can control the generation of the Y type.

The propagation of the Y TYPE is specified bit-by-bit for each of the eight type bits. Each resulting type can be directly specified as 0 or 1, or inherited from a bit in the same position from either the A or B input types. For each bit position $i = 0, \ldots, 7$ consider the two controls, m_i and v_i, encoded as follows:

Type Propagation Controls

m_i	v_i	Result TYPE$_i$
0	0	0
0	1	1
1	0	A TYPE$_i$
1	1	B TYPE$_i$

The TMASK field is a set of eight m_i and v_i controls as follows:

TMASK	
$m_7, \ldots, m_0$ 8	$v_7, \ldots, v_0$ 8

TMASK is used to generate the Y TYPE *except* when otherwise specified by a UNIT = 10 OP as follows:

Type Operations UNIT = 10			
OP	Operation	Y TYPE	Y IMMEDIATE
0	SETTYPE	A	A TYPE
1	GETTYPE	TMASK	A TYPE

Note that the type specification for SETTYPE, A, is in the same format as TMASK so SETTYPE can be used to set or reset individual type bits.

A.2.5 Machine control unit

The machine control unit (MCU) permits program access of machine control settings. There are three classes of controls, (1) stack control, (2) exception control and (3) statistics. The stack controls permit the setting of stack bases and current top of stack for the two token queues. The token queues are cached from the local memory. It is also possible to inhibit popping from a stack (NOPOP) and to swap the relative stack priorities (STACKSWAP). These are used to perform ideal time step simulations and allow an operating system to occasionally "flush out" a stack.

MCU Stack Controls UNIT = 11		
OP	Operation	Description
1 000 0000	SETBASE0	*stack*0 base ← A
1 000 0001	SETBASE1	*stack*1 base ← A
1 000 0010	SETTOS0	*stack*0 TOS ← A
1 000 0011	SETTOS1	*stack*1 TOS ← A
1 000 0100	SETNOPOP0	NOPOP0 ← A
1 000 0101	SETNOPOP1	NOPOP1 ← A
1 000 0110	SETSWAP	STACKSWAP ← A
0 000 0000	GETBASE0	Y ← *stack*0 base
0 000 0001	GETBASE1	Y ← *stack*1 base
0 000 0010	GETTOS0	Y ← *stack*0 TOS
0 000 0011	GETTOS1	Y ← *stack*1 TOS
0 000 0100	GETNOPOP0	Y ← NOPOP0
0 000 0101	GETNOPOP1	Y ← NOPOP1
0 000 0110	GETSWAP	Y ←STACKSWAP

The exception controls provide access to the saved exception state register comprising the left and right operands and an exception status word. When an exception occurs an exception flag is set which must be cleared before another exception can be processed correctly. There are actually six sets of exception state registers, one for each pipeline interleave. The MCU instructions implicitly address the correct set.

MCU Exception Controls UNIT = 11		
OP	Operation	Description
0 001 0000	GETA	$Y \leftarrow A_{exception}$
0 001 0001	GETB	$Y \leftarrow B_{exception}$
0 001 0010	GETSTATUS	$Y \leftarrow$ status word
1 001 0011	CLEAR	Clear exception flag

A.2.6 Next Address Control

The next address control field NACTL specifies increments to the current tag IP to generate two new destination tags, *tag*1 and *tag*2. NACTL has two subfields:

NACTL	
NA1	NA2
2	1

where,

NA1 Specifies the *tag*1 IP increment of 0,1,2, or 3. PORT is always set to *l*.

NA2 Specifies the *tag*2 IP increment of 0 or *s*. PORT is set to the instruction PORT or *r*.

Encoding of NA1		
NA1	*tag*1 IP	*tag*1 PORT
00	IP	*l*
01	IP + 1	*l*
10	IP + 2	*l*
11	(IP OR 1) + 3	*l*

Encoding of NA2		
NA2	*tag*2 IP	*tag*2 PORT
0	IP	*r*
1	IP + *s*	inst PORT

Note that NA1 = 0, NA2 = 0 produces tags that refer to the left and right ports of the *current* instruction. The last option on NA1 (11) is used to skip a literal value that occupies the next word in memory. Because instructions are halfword aligned (two to a word) IP must first be rounded to a consistent boundary.

A.2.7 Form Token Section

The form token section assembles zero, one, or two output tokens from the function unit outputs Y and B and the next address outputs *tag*1 and *tag*2. The unit also controls the token stacks and automatically forwards tokens to the network.

A.2.8 Form Token Control

The FTCTL field has the following structure:

FTCTL							
EN1	EN2	K1	K2	ORD	RECIRC	STACK	ACK
2	2	2	2	1	2	1	1

where,

EN1, EN2 Specifies the conditional output predicates for *token*1 and *token*2.

K1, K2 Specifies the assembly of *token*1 and *token*2.

ORD Specifies the relative priority of *token*1 and *token*2.

RECIRC Controls the recirculation of the higher priority token.

STACK Specifies the stack for the lower priority token.

ACK Controls acknowledgment for network packets.

Two logical tokens *token*1 and *token*2 are generated from EN1, K1 and EN2, K2 specifications. In the case of two tokens being generated ORD designates *token*1 or *token*2 as the highest priority token, *i.e.*, the token that will be recirculated. RECIRC controls the recirculation of this token. In the case that only one token is formed, or in the case that two are formed and the other is sent over the network, the token recirculation is controlled by RECIRC. In the event that two tokens are formed and both remain local to PE then lower priority token is stacked on stack STACK.

Encoding of EN1 and EN2

EN_i	$token_i$ is generated on ALU condition
00	always
01	$= 0$
10	never
11	$\neq 0$

K1 specifies the construction of *token*1 as follows:

Encoding of K1

K1	*token*1 TAG	*token*1 VALUE
00	*tag*1	*Y*
01	*tag*1	*B*
10	*Y*	*tag*1
11	*B*	*Y*

K2 specifies the construction of *token*2 as follows:

Encoding of K2

K2	*token*2 TAG	*token*2 VALUE
00	*tag*2	*Y*
01	*Y*	*B*
10	*tag*1	*tag*2
11	*Y*	*tag*2

ORD specifies the relative ranking of *token*1 and *token*2 in the case that both are generated and both remain local to PE.

Encoding of ORD

ORD	Recirculated	Stacked
0	*token*1	*token*2
1	*token*2	*token*1

A single token or the higher priority token, as defined by ORD is controlled by RECIRC.

Encoding of RECIRC

RECIRC	Action
00	Normal recirculation
01	Uninterruptible recirculation
10	Push on *stack*0
11	Push on *stack*1

In the case of two tokens, the lower priority token is *always* stacked even if RECIRC designates stacking of the higher priority token. If both RECIRC and STACK specify the same stack then the lower priority token is pushed first.

Encoding of STACK	
STACK	Action
0	Push on *stack*0
1	Push on *stack*1

Finally, the form token section may demand a positive (*i.e.*, circuit-switched) acknowledgment for a network-bound token.

Encoding of ACK	
ACK	Action
0	Immediately process local token
1	Release local token only after network ack

The local token *must* be designated as low priority *stack*1. The hardware inhibits release of the local token by inhibiting *stack*1 pops until *all* outstanding acknowledgments have been received.

A.2.9 Stack Management

There are two token stacks, *stack*0 and *stack*1, where *stack*0 has priority over *stack*1. That is, *stack*1 will be popped only if *stack*0 is empty.

In the case of a single processor without a network connection there are three possible results of the form token section:

Form Token Results Without Network	
Form Token Output	Action
no tokens	if *stack*0 not empty then pop *stack*0 else if *stack*1 not empty then pop *stack*1 else insert idle token (IP = 0).
*k*1 or *k*2	if RECIRC = 0X then recirculate token else push on RECIRC stack and insert idle token.

*k*1 and *k*2	if RECIRC = 0X then recirculate ORD token, push other
	else push non-ORD then push ORD, insert idle.

Note that if the MCU control bits NOPOP0 or NOPOP1 are set then, for purposes of the above algorithm, the associated stack always tests empty. If the MCU STACKSWAP bit is set then roles of *stack*0 and *stack*1 are reversed.

When a network is involved things get a little more complex. An incoming message has priority over everything except an uninterruptible recirculating token. If the form token section produces no tokens, then the incoming token is recirculated (instead of either popping a stack or inserting an idle, as above). If one token is produced, and it is interruptible, then the token is pushed (on *stack*0 by default) and the incoming message is recirculated. If two tokens are produced then both are pushed (first the low priority then the high priority, again the high priority defaulting to *stack*0) and the incoming message is recirculated. If a produced token is uninterruptible then the incoming message is blocked until the next cycle. *Incoming network tokens are never pushed onto a stack and are always processed in FIFO order.*

Outgoing messages are routed directly to the network, but to avoid deadlock, they will be pushed onto *stack*0 if the outgoing network connection is presently blocked. This implies that a token popped from a stack may head for the network rather than be recirculated.

A.3 Exceptions

Exceptions can be generated from a masked set of function unit status outputs. When an exception is detected, the current A and B values, along with the status outputs, are recorded into temporary registers that are accessible through the MCU. An exception handler is invoked and is passed the TAG of the offending activity. Exception handlers are entered as critical sections (RECIRC = 01) so there is the possibility that there will be as many exception handlers active simultaneously as there are pipeline stages[1]. The hardware sorts this out by maintaining six distinct contexts for the faulting A and B values and transparently provides the correct value for each active exception.

Similarly, the system programmer must reserve eight activation frames, one for each possible pipeline thread. Aside from these contexts there is no hardware vectoring for exceptions. So the first thing an exception handler must do is determine the exception class by fetching the faulting instruction and examining the saved status bits. The exception mask EMASK has the following structure:

EMASK									
SENSE	ALWAYS	DIVZ	UF	OF	INX	NaN	DEN	ZERO	NEG
1	1	1	1	1	1	1	1	1	1

[1] If an exception occurs within an exception handler before it clears the exception flag in the MCU, an unrecoverable machine check occurs.

where,

SENSE Exception occurs when any unmasked bit is asserted (SENSE = 0) or when no unmasked bit is asserted (SENSE = 1).

ALWAYS Always asserted.

DIVZ Asserted when FPU detects divide by zero.

UF Asserted when FPU or PIU detects an underflow.

OF Asserted when FPU or PIU detects an overflow.

INX Asserted when FPU result is inexact.

NaN Asserted when an FPU input is not a number.

DEN Asserted when an FPU input is denormalized.

ZERO Asserted when FPU result is zero.

NEG Asserted when FPU result is negative.

An exception is masked when the corresponding **EMASK** bit is set to zero. A masked version of the status word is recorded in the MCU and can be retrieved by a GETSTATUS operation. Similarly, the associated A and B values can be obtained by an MCU GETA and GETB. The exception handler is invoked with a TAG that has FP $= 0$ and an IP $= n+1$ where n is the logical pipeline thread that received the exception. The VALUE part contains the TAG that caused the exception.

A.4 Statistics

The hardware monitoring function consists of a set of sixteen 32 bit counters. The STATS field selects one of the counters and increments it. The host processor or MCU can read and/or reset any of the counters.

MCU Statistics Controls UNIT = 11		
OP	Operation	Description
0 010 0000	ACTIVITY?	$Y \leftarrow$ *activity flag*; *activity flag* $\leftarrow 0$
0 010 0001	GETCOUNTER	$Y \leftarrow counter(A)$
1 010 0001	SETCOUNTER	$counter(A) \leftarrow B$

Note that the *activity flag* is automatically set by any instruction that does not explicitly clear it. Idle instructions should always clear this flag. Another processor can read this flag (by sending a token into the processor with its return tag—just like an **i-fetch**) to determine if the processor is idle. This can be used for ideal time step simulations to establish when the current step is complete.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison–Wesley, Reading MA, 1986.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An Overview of the PTRAN Analyzer System for Multiprocessing. In *Proceedings of the International Conference on Supercomputing, Athens, Greece*, 1987.

[3] J.R. Allen and K. Kennedy. PFC: A Program to Convert Fortran to Parallel Form. In *Proceedings of the IBM Conference on Parallel Computers and Scientific Computations*, March 1982.

[4] Arvind, S. A. Brobst, and G. K. Maa. Evaluation of the MIT Tagged-Token Dataflow Project. Technical Report CSG Memo 281, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1988.

[5] Arvind and D. E. Culler. Managing Resources in a Parallel Machine. In *Proceedings of IFIP TC-10 Work rg Conference on Fifth Generation Computer Architecture, Manchester, England.* North-Holland Publishing Company, July 15–18 1985.

[6] Arvind and D. E. Culler. Dataflow Architectures. In *Annual Reviews in Computer Science*, volume 1, pages 225–253. Annual Reviews Inc., Palo Alto, CA, 1986.

[7] Arvind and K. Ekanadham. Future Scientific Programming on Parallel Machines. In *Proceedings of the International Conference on Supercomputing, Athens, Greece*, June 8–12 1987.

[8] Arvind, K. Ekanadham, and D. E. Culler. The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures. In *CONPAR 88, Manchester, England*, 1988.

[9] Arvind and K.P. Gostelow. The U-interpreter. *Computer*, 15(2), February 1982.

[10] Arvind, K.P. Gostelow, and Plouffe W. An Asynchronous Programming Language and Computing Machine. Technical Report TR–114a, University of California, Irvine, Depatment of Information and Computer Science, Irvine, CA, December 1978.

[11] Arvind and R. A. Iannucci. Instruction Set Definition for a Tagged-Token Dataflow Machine. Technical Report CSG Memo 212-3, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, February 1983.

[12] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proceedings of DFVLR – Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany*, June 25–29 1987.

[13] Arvind, G. K. Maa, and D. E. Culler. Parallelism in Dataflow Programs. In *Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, May 30–June 2 1988.

[14] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands. (Lecture Notes in Computer Science Volume 259)*. Springer-Verlag, June 15–19 1987.

[15] J. Backus. Can Programming be Liberated from the von Neumann Style? *Communications of the ACM*, 21(8), August 1978.

[16] S.A. Brobst. Instruction Scheduling and Token Storage Requirements in a Dataflow Supercomputer. Technical report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, May 1986.

[17] D. E. Culler. *Effective Dataflow Execution of Scientific Programs.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, December 1988 (expected).

[18] D. E. Culler and Arvind. Resource Requirements of Dataflow Programs. In *Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, May 30–June 2 1988.

[19] J. B. Dennis. Data Flow Supercomputers. *Computer*, 13(11), November 1980.

[20] K. Ekanadham, Arvind, and D. E. Culler. The Price of Parallelism. In *Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, May 30–June 2 1988.

[21] D.D. Gajski, D.A. Padua, D. J. Kuck, and R.H. Kuhn. A Second Opinion on Dataflow Machines and Languages. *Computer*, 15(2):58–69, February 1982.

[22] J. R. Gurd, C.C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[23] R. H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[24] R. H. Halstead and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proceedings of the 15th International Symposium on Computer Architecture*, 1988.

[25] S.K. Heller. An I-Structure Memory Controller (ISMC). Technical Report CSG Memo 239, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, May 1984.

[26] K. Hiraki, S. Sekiguchi, and T. Shimada. System Architecture of a Dataflow Supercomputer. Technical report, Computer Systems Division, Electrotechnical Laboratory, 1-1-4 Umezono, Sakura-mura, Niihari-gun, Ibaraki, 305, Japan, 1987.

[27] R. W. Hockney. Classification and Evaluation of Parallel Computer Systems. In *Parallel Computing in Science and Engineering*. Springer-Verlag, 1988.

[28] R. A. Iannucci. A Dataflow/von Neumann Hybrid Architecture. Technical Report LCS TR–418, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, May 1988.

[29] Dongarra J. J. Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment. *ACM SIGARCH Computer Architecture News*, 16(1), March 1988.

[30] A. H. Karp. Programming for Parallelism. *Computer*, 20(5), May 1987.

[31] Vinod Kathail. Addendum to the Instruction Set Definition for a Tagged-Token Dataflow Machine. Technical Report CSG Memo 212–3–1, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, October 1983.

[32] D. Kuck, E. Davidson, D. Lawrie, and A. Sameh. Parallel Supercomputing Today and the Cedar Approach. *Science Magazine*, 231:967–974, February 1986.

[33] O. M. Lubeck. Supercomputer Performance: The Theory, Practice, and Results. Technical Report LA–11204–MS, Los Alamos National Laboratory, January 1988.

[34] G. Maa. Code-Mapping Policies for the TTDA. Master's thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, December 1987.

[35] D. A. Mandell. Experience and Results Multitasking a Hydrodynamics Code on Global and Local Memory Machines. In *International Conference on Parallel Processing*, August 1987.

[36] D. A. Mandell and H. E. Trease. Parallel Processing a Real Code – A Case History. Technical Report LA–UR–88–1836, Los Alamos National Laboratory, May 1988.

[37] J. R. McGraw. SISAL: Streams and Iteration in a Single Assignement Language, Language Reference Manual, Version 1.2. Technical Report M–146, Lawrence Livermore National Laboratory, March 1985.

[38] NEC. *Advanced Product Information Users Manual: uPD7281 Image Pipelined Processor.* NEC Electronics, Inc., Mountain View, CA, 1985.

[39] R. S. Nikhil. Id Nouveau Reference Manual, Part I: Syntax. Technical report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[40] R. S. Nikhil. Id World Reference Manual. Technical report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[41] R. S. Nikhil and Arvind. A von Neumann/Dataflow Abstract Machine Language. Technical Report Notes for Course 6.823s, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, July 1988.

[42] G. F. Pfister *et al.* The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, 1985.

[43] S. Plotkin. Uncompleted Manuscript, 1986.

[44] McGraw J. R. and Axelrod T. S. Exploiting Multiprocessors: Issues and Options. In *Programming Parallel Processors.* Addison-Wesley, Reading MA, 1988.

[45] R. Rettberg *et al.* Development of a Voice Funnel System. Technical Report Design Report 4098, Bolt Beranek and Newman Inc., Cambridge, MA, August 1979.

[46] V. Sarkar. Paritioning and Scheduling Parallel Programs for Execution on Multiprocessors. Technical Report CSL–TR–87–328, Stanford University Computer Systems Laboratory, April 1987.

[47] C.L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1), January 1985.

[48] B. J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8, 1978.

[49] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR–370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986.

OFFICIAL DISTRIBUTION LIST

Director 2 copies
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Office of Naval Research 2 copies
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. R. Grafton, Code 433

Director, Code 2627 6 copies
Naval Research Laboratory
Washington, DC 20375

Defense Technical Information Center 12 copies
Cameron Station
Alexandria, VA 22314

National Science Foundation 2 copies
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

Dr. E.B. Royce, Code 38 1 copy
Head, Research Department
Naval Weapons Center
China Lake, CA 93555

END

6-89

DTIC